



Università degli Studi di Parma

Dipartimento di Ingegneria dell'Informazione

Sistemi operativi e in tempo reale - a.a. 2022/23

Sistemi in tempo reale

Definizioni

prof. Stefano Caselli



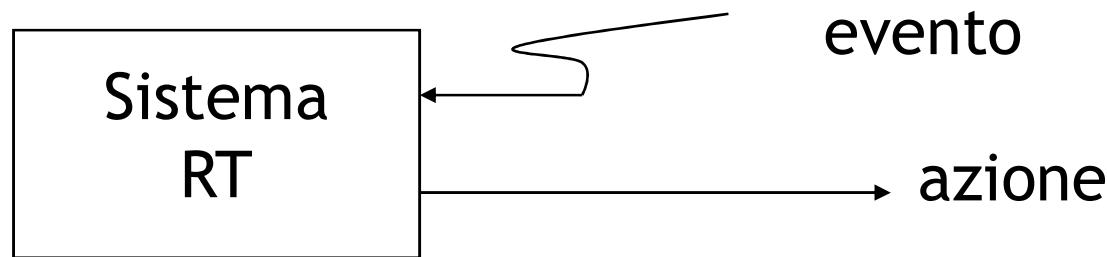
Esempi discussi

-
- Il generico sistema di controllo digitale multirate
 - Il sistema avionico
 - Il controllo di un robot mobile in ambiente domestico
 - ...

 - Q: Cosa succede in questi sistemi se non si reagisce *in tempo utile* agli eventi?



Sistema in tempo reale

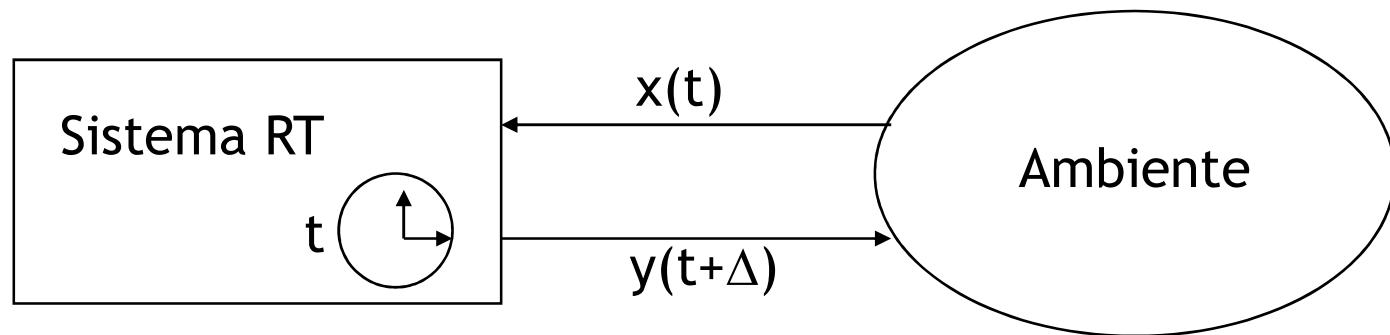


- Un *Sistema in Tempo Reale* è un sistema di elaborazione in grado di rispondere agli eventi rispettando precisi vincoli temporali



Sistema in tempo reale - definizione

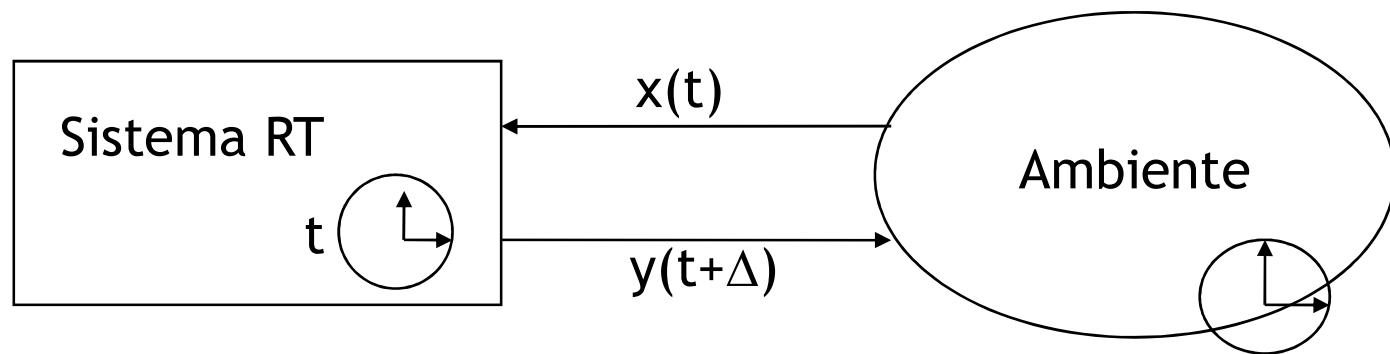
- Un sistema in tempo reale è un sistema in cui la *correttezza* della elaborazione dipende *non solo dai risultati* prodotti in uscita, *ma anche dall'istante* in cui tali risultati vengono resi disponibili





Sistema in tempo reale

- Il tempo del sistema deve essere sincronizzato con il tempo dell'ambiente

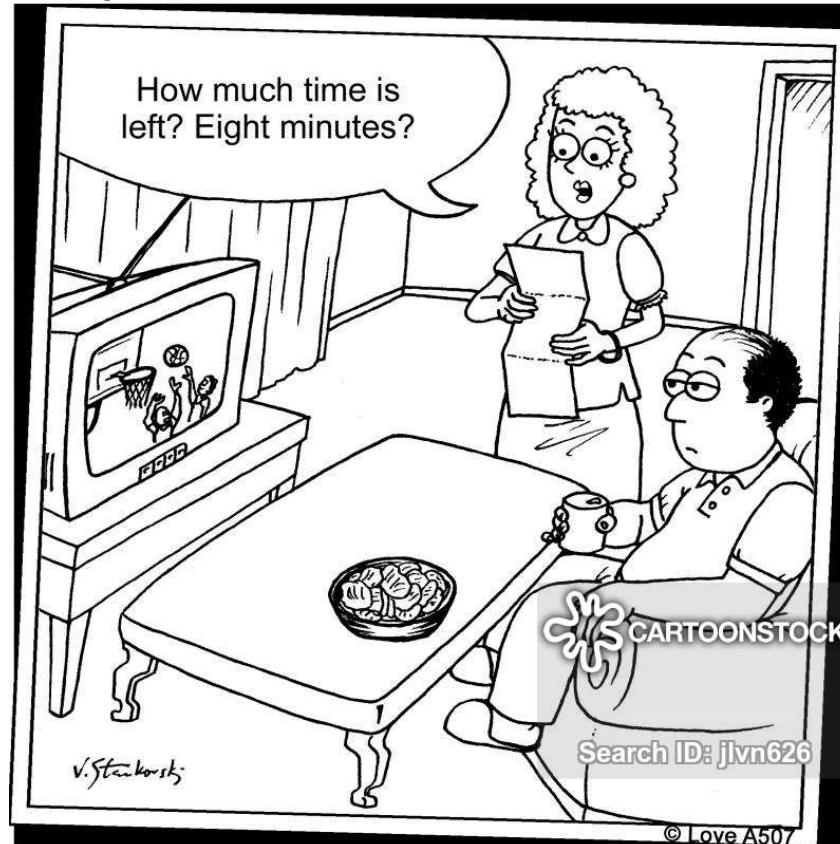


- In un sistema non in tempo reale manca questa dipendenza da vincoli temporali



Altre definizioni di real-time

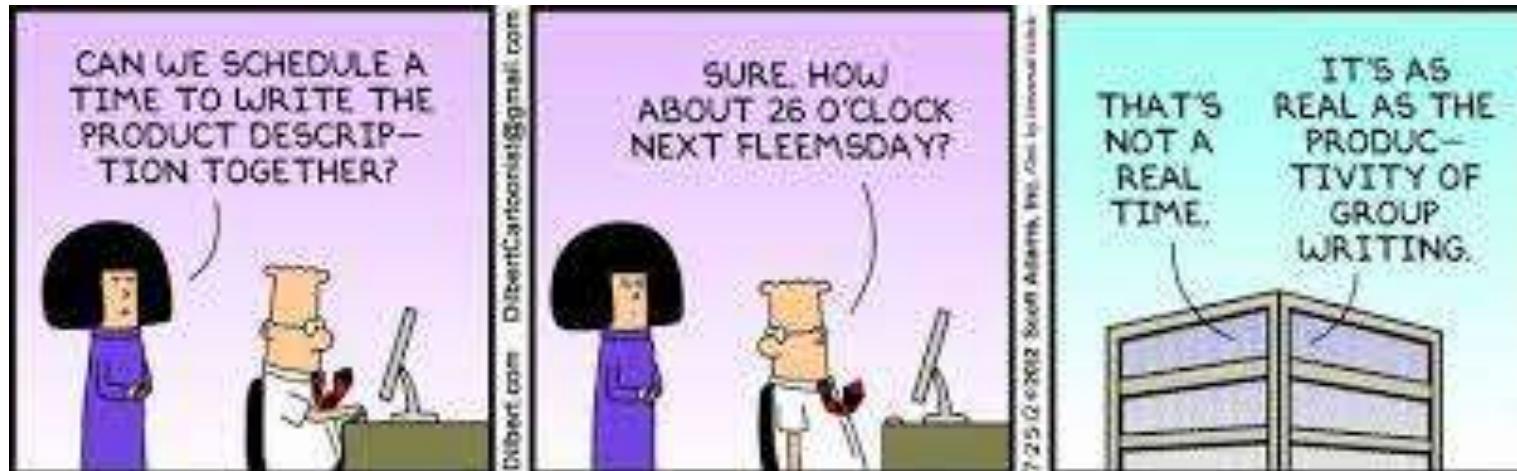
Snapshots



Shelly consults her basketball-time-to-real-time conversion chart.



Altre definizioni di real-time





Velocità

- Real-time \neq veloce!
- La velocità è sempre relativa a quella dello specifico ambiente in cui opera il sistema
- Un'esecuzione più veloce è in genere utile, ma non *garantisce* di per sé un comportamento corretto



Velocità

-
- L'obiettivo di un sistema real-time è garantire il *comportamento temporale* di ciascun task
 - L'obiettivo di un sistema veloce è minimizzare il *tempo medio di risposta* di un *insieme di task*

 - I tempi medi tuttavia non garantiscono le prestazioni dei singoli task!



Predicibilità

- Requisito essenziale di un sistema RT è la *predicibilità*
- In presenza di elevate incertezze (sulle caratteristiche del task set e del sistema di elaborazione) è impossibile o difficile fornire garanzie
- La predicibilità al 100% esiste nelle analisi formali, mentre si trova di rado nei problemi reali
 - *analisi di caso peggiore* e inclusione degli *overhead*
- Se la variabilità è eccessiva, si può valutare se è comunque possibile e utile fornire *garanzie ad un sottoinsieme di task*



Fonti di non determinismo

- Architettura
 - cache, elaborazione in pipeline, esecuzione speculativa, branch prediction, interrupt, DMA
 - → trend verso architetture ad alte prestazioni ma con crescente non determinismo
- Sistema operativo
 - scheduling, sincronizzazione, comunicazione, memoria virtuale
- Linguaggio di programmazione
 - possibile assenza di supporti per la gestione del tempo
- Metodologia di progetto
 - assenza di tecniche di analisi e verifica



Definizioni (Liu)

- *Job*: unità di lavoro schedulata ed eseguita dal sistema
- *Task*: insieme di job correlati che realizzano collettivamente una funzionalità del sistema
- *Processore*: risorsa necessaria al job per l'esecuzione (CPU, disco, rete);
 - *server* in teoria delle code
- Astraiamo, in generale, dalla specifica elaborazione o trasmissione (job) e dalla specifica risorsa (processore)



Vincoli temporali

- *Istante di rilascio* (istante di attivazione, istante di richiesta): istante in cui il job diviene disponibile per l'esecuzione
- *Deadline* (scadenza): istante entro cui l'esecuzione deve essere completata
 - *Deadline relativa*: massimo tempo di risposta tra l'istante di rilascio e l'istante di completamento
 - *Deadline assoluta* (= Deadline):
= istante di rilascio + deadline relativa
- Istante di rilascio e deadline consentono di specificare i vincoli temporali più frequenti

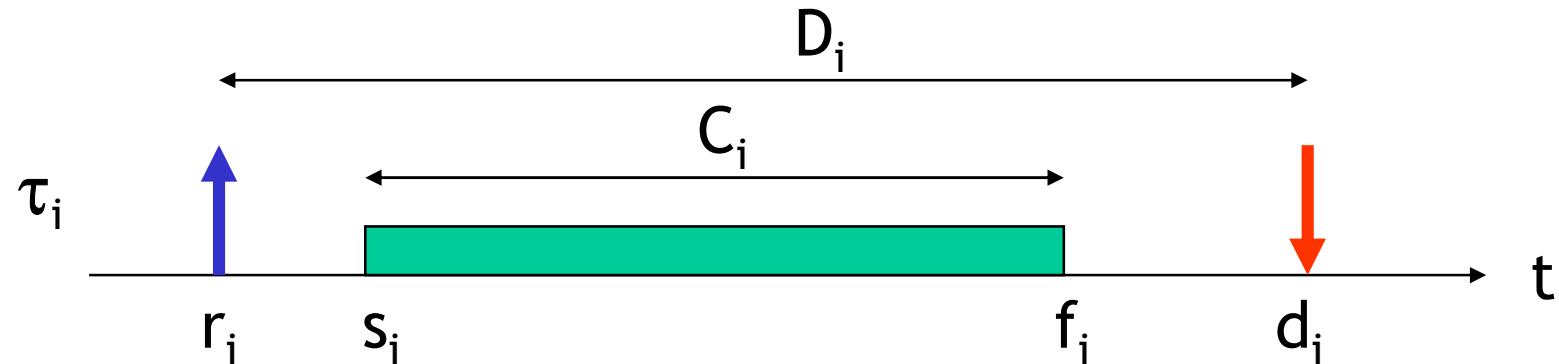


Deadline *hard* e *soft*

- *Deadline miss*: mancato rispetto di una deadline (produzione in ritardo del risultato, ovvero il risultato non è pronto nell'istante di deadline)
- Intuitivamente:
 - *Hard deadline*: una deadline miss è considerata un *guasto fatale* o che può provocare conseguenze disastrose
 - *Soft deadline*: una deadline miss è *indesiderabile*, ma tollerabile se non troppo frequente; miss ripetute rendono le prestazioni del sistema sempre più scadenti
- Problema: Definizioni di *deadline hard* e *soft* non formalizzate né riferite a valori quantificabili!



Parametri dei job real-time



r_i istante di rilascio (o di richiesta, tempo di arrivo a_i)

s_i istante di inizio (*start time*)

C_i tempo di esecuzione del job (*execution time*)

$WCET_i$ (*worst-case execution time*) valore max che può assumere C_i

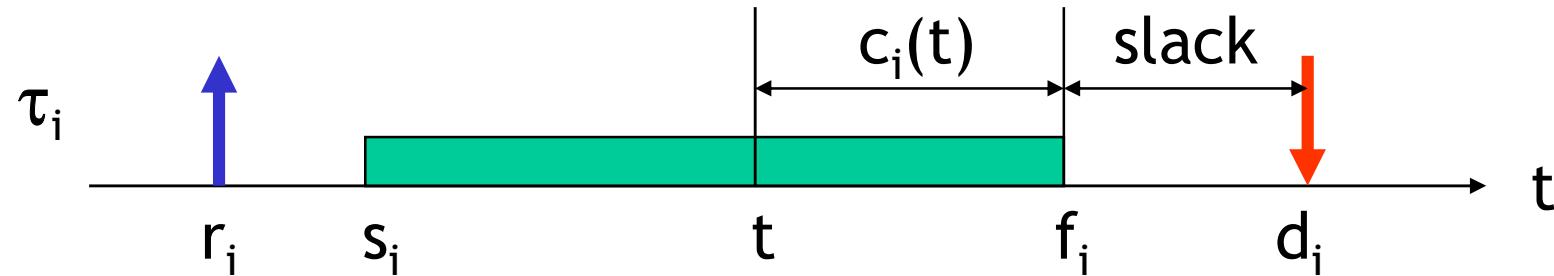
d_i deadline assoluta (*scadenza*)

D_i deadline relativa

f_i ist. di completamento (*completion o finishing time*)



Altri parametri



Lateness: $L_i = f_i - d_i$

Tardiness: $\max(0, L_i)$

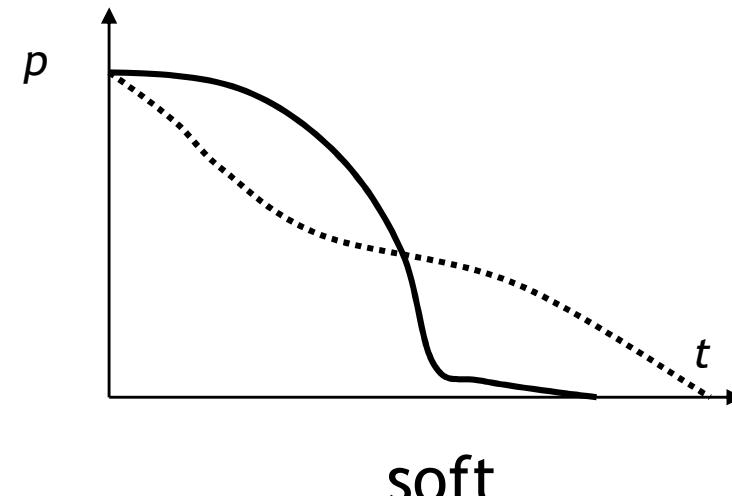
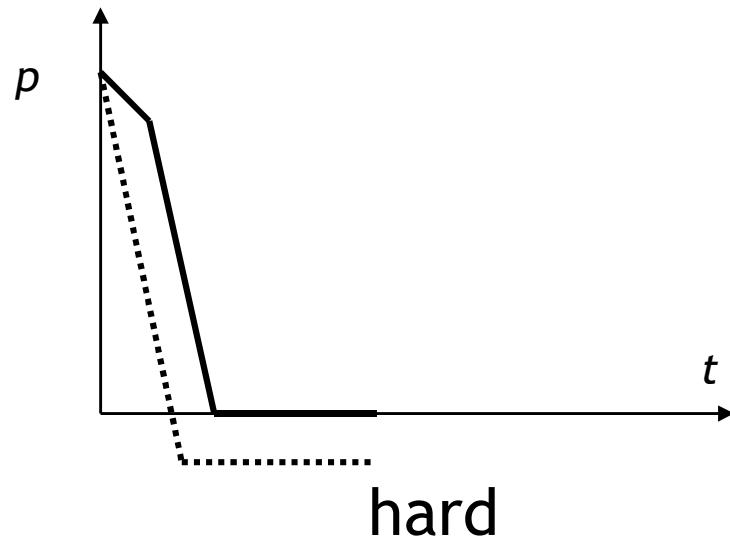
WCET residuo: $c_i(t), c_i(r_i) = C_i$

Lassità (slack): $d_i - t - c_i(t)$



Deadline *hard* e *soft*

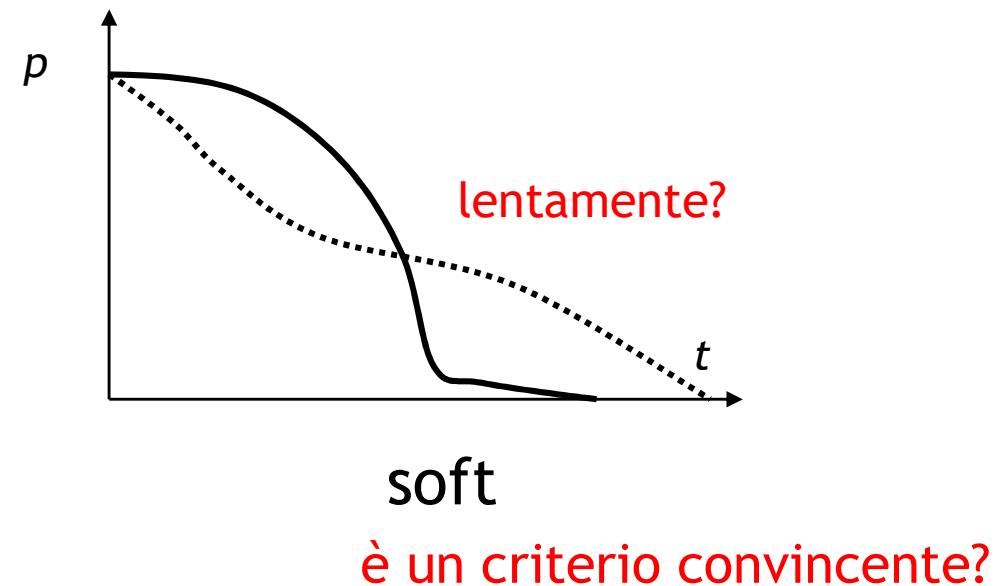
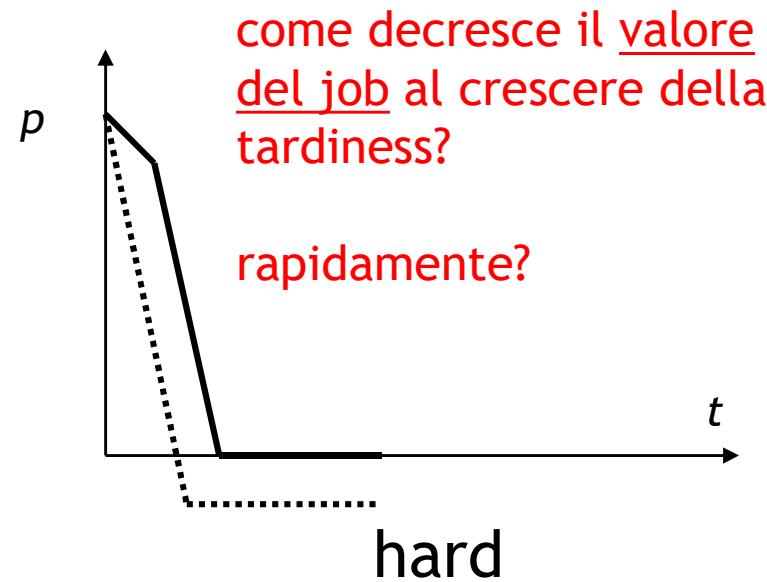
- Misura quantitativa: *prestazione complessiva* p del sistema in funzione del ritardo dei job (*tardiness*, t)
- Tardiness: ritardo di completamento rispetto alla deadline, è 0 se il job completa entro la deadline





Deadline *hard* e *soft*

- Misura quantitativa: *prestazione complessiva* p del sistema in funzione del ritardo dei job (*tardiness*, t)
- Tardiness: ritardo di completamento rispetto alla deadline, è 0 se il job completa entro la deadline





Deadline *hard* e *soft*

- Anche una definizione basata sul concetto di valore in funzione della tardiness $p(t)$ non è risolutiva, per la difficoltà di stabilire metriche univoche
- Definizione *operativa*:

Un *job* ha una deadline di tipo *hard* quando il progettista deve *dimostrare* che il *job* *rispetta sempre la deadline* nelle condizioni di funzionamento specificate



Sistemi hard real-time

- Un *sistema* real-time viene definito *hard real-time* se *alcuni dei job* da cui è composto presentano *deadline di tipo hard*
 - Nota: almeno due job hard... garantire un job è banale!
- Esempi:
 - Sistemi embedded
 - Procedure di recovery in sistemi high-availability
- I sistemi hard real-time che devono gestire *dinamiche di impianto molto veloci* tipicamente richiedono *sistemi operativi specializzati* (esigenze di determinismo e vincoli di latenza max)

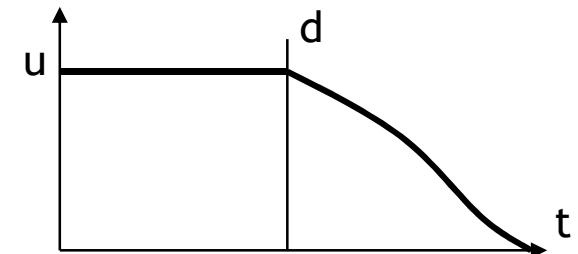


Sistemi soft real-time

- Un sistema real-time viene definito *soft real-time* se i job hanno deadline di tipo soft

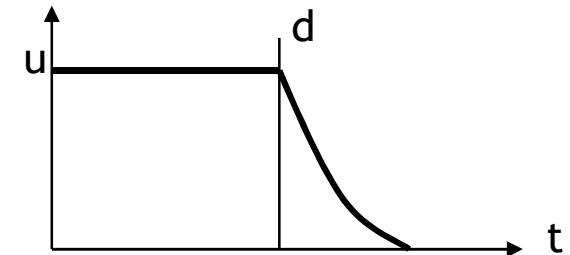
Vincoli temporali non stringenti:

- sistemi per transazioni on-line
- centraline di commutazione



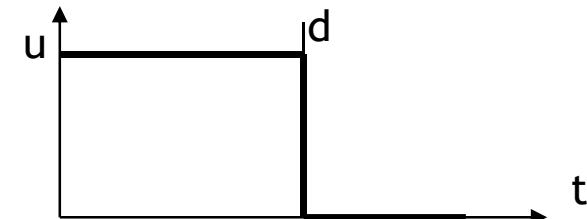
Vincoli temporali più stringenti:

- sistema di borsa telematica



Vincoli temporali stringenti:

- multimedia





Sistemi soft real-time

-
- I *requisiti* sono spesso specificati in termini probabilistici
 - La *validazione* avviene di solito mediante simulazione e prove sul sistema



Richiami e notazioni

- Da SisOp...
- Liu utilizza il termine *Job*; nella letteratura real-time vengono tuttavia utilizzati anche i termini *Task*, *Processo*, *Thread*
- Usiamo i termini Job, Task, Processo in modo intercambiabile, ove non sorga confusione
- Usiamo Thread in contrapposizione a Processo
- Più precisamente: il Thread è il *meccanismo* del sistema operativo per eseguire l'*attività*, Task o Job, oggetto dello scheduling in tempo reale



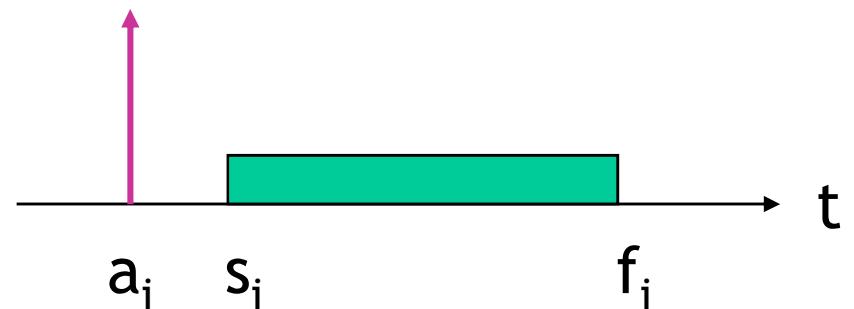
Thread non real-time

- Thread o task:
sequenza di istruzioni che, in assenza di altre attività, viene eseguita in modo continuativo dal processore fino al suo completamento

a_i = tempo di arrivo
(arrival time)

s_i = tempo di inizio esec.
(start time)

f_i = tempo di fine esec.
(finishing time)



Cosa manca in questa figura?



Stato di un thread

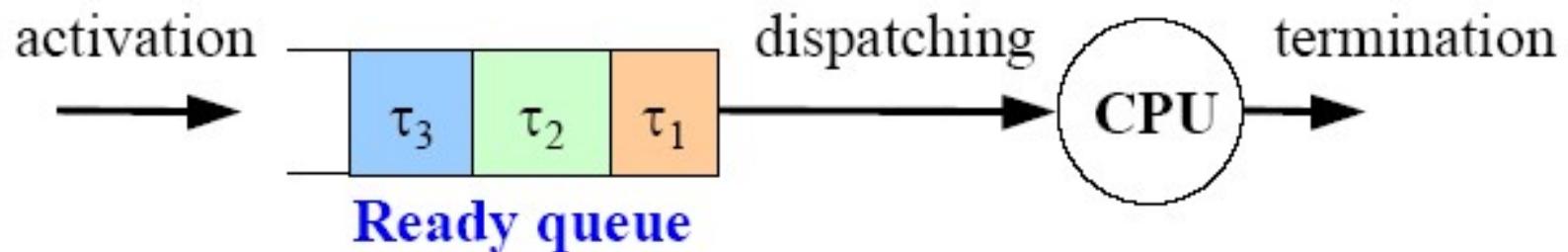
- Prescindendo dalla disponibilità della CPU:
 - *Attivo*: il task può essere eseguito dalla CPU
 - *Bloccato*: in attesa di un evento

- Un task *Attivo* può essere:
 - *Running*: in esecuzione dalla CPU
 - *Pronto*: in attesa della CPU



I thread pronti

- I descrittori dei thread pronti sono organizzati in una coda di attesa (o altra struttura dati più generale) denominata *ready queue* (coda dei thread pronti)
- La strategia per la scelta del task da porre in esecuzione sulla CPU è l'*algoritmo di scheduling*





Scheduling e revoca

- **Scheduling *preemptive***

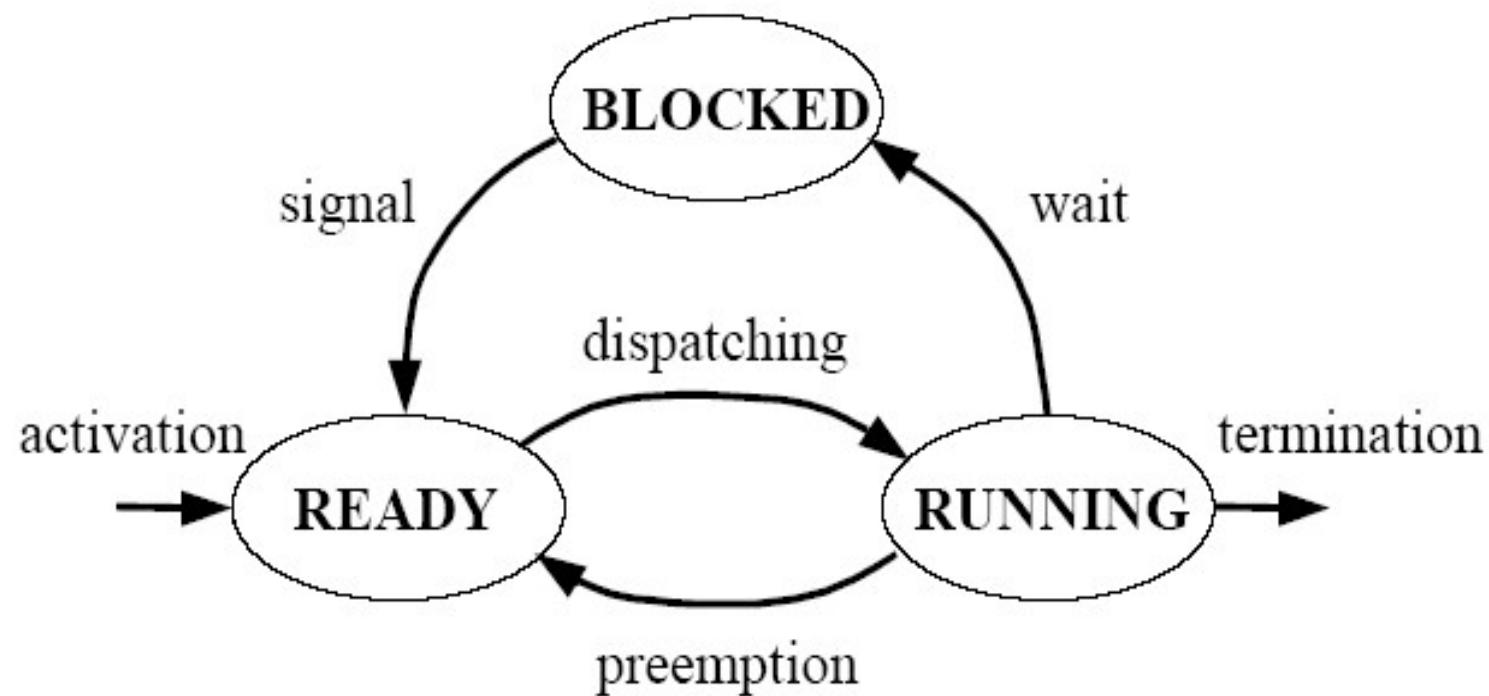
il task in esecuzione può essere temporaneamente sospeso, inserendolo nella ready queue, a favore di un task più importante

- **Scheduling *non preemptive***

il task in esecuzione non è soggetto a revoca fino al suo completamento



Transizioni di stato dei task





Schedule

- Una schedule (lista di assegnamento) è uno specifico assegnamento di task al processore
- Dato un insieme di task $\Gamma = \{\tau_1, \dots, \tau_n\}$, una schedule è un mapping $\sigma : R^+ \rightarrow N$ tale che $\forall t \in R^+, \exists t_1, t_2 : t \in [t_1, t_2)$ e $\forall t' \in [t_1, t_2) : \sigma(t) = \sigma(t')$

$\sigma(t) = k > 0$ se τ_k è in esecuzione

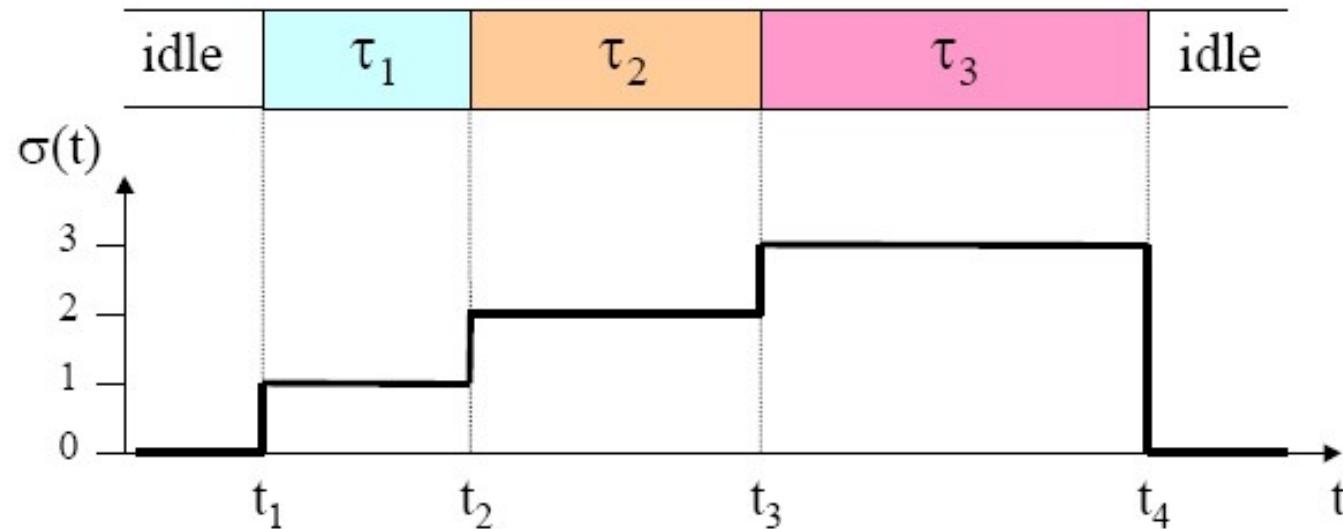
$\sigma(t) = 0$ se il processore è inattivo (idle)

THINGS TO DO:
◀◀◀ ▶▶▶

<input type="checkbox"/>	_____



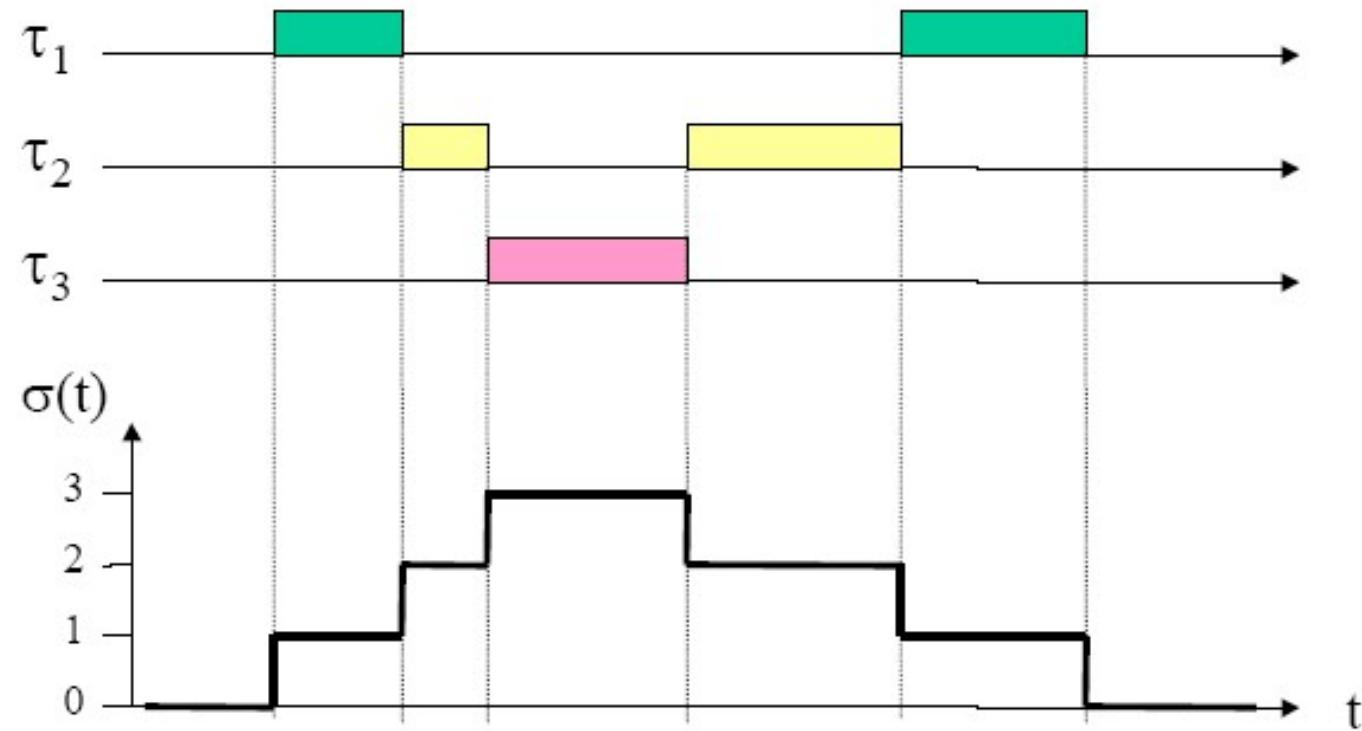
Un esempio di schedule



- Agli istanti t_1 , t_2 , t_3 , t_4 viene eseguito un *thread switch*
- Ogni intervallo $[t_i, t_{i+1})$ è denominato *time slice*



Schedule con preemption





Università degli Studi di Parma

Dipartimento di Ingegneria dell'Informazione

Sistemi operativi e in tempo reale - a.a. 2022/23

Modello di riferimento per sistemi real-time

prof. Stefano Caselli

stefano.caselli@unipr.it

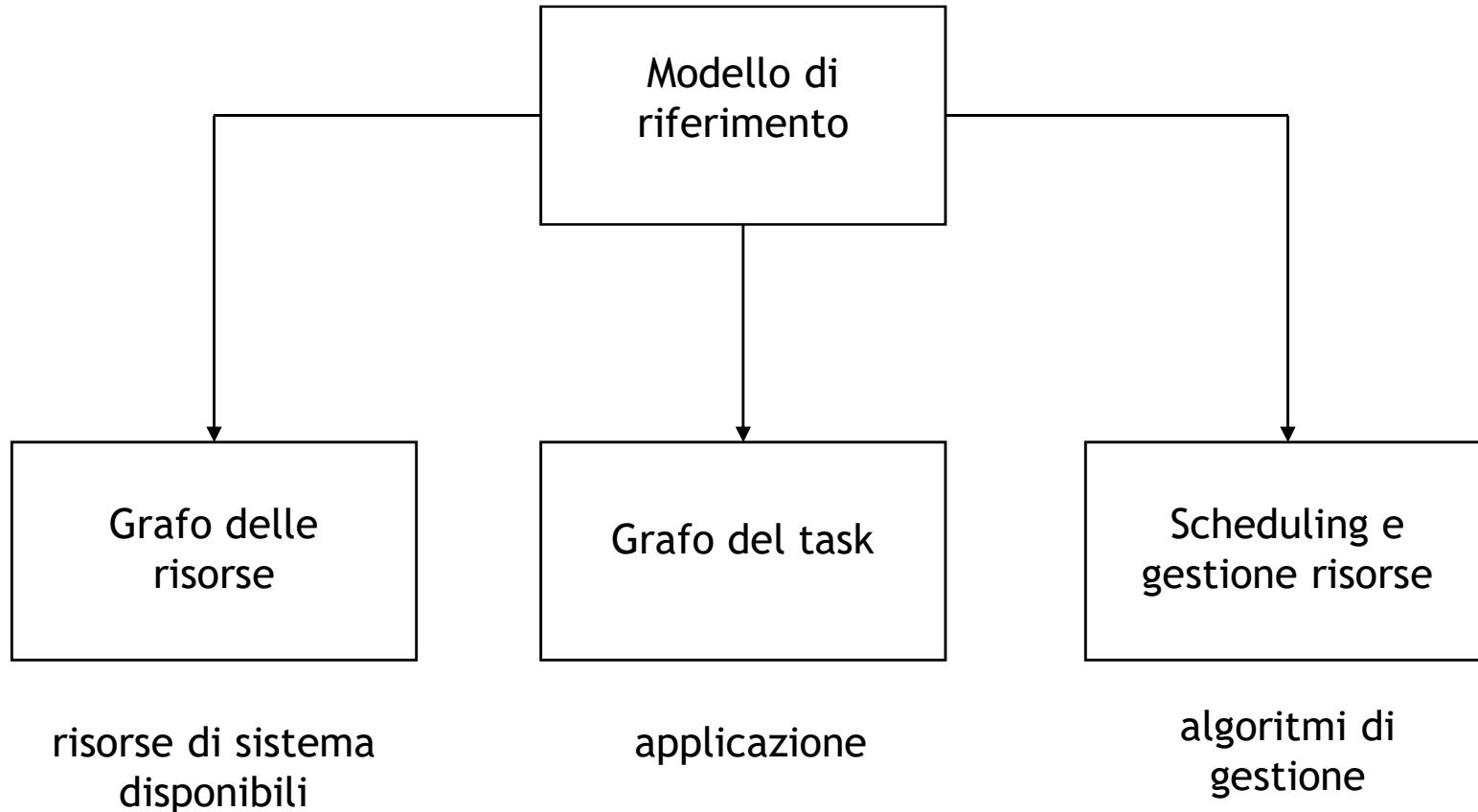
<http://rimlab.ce.unipr.it>



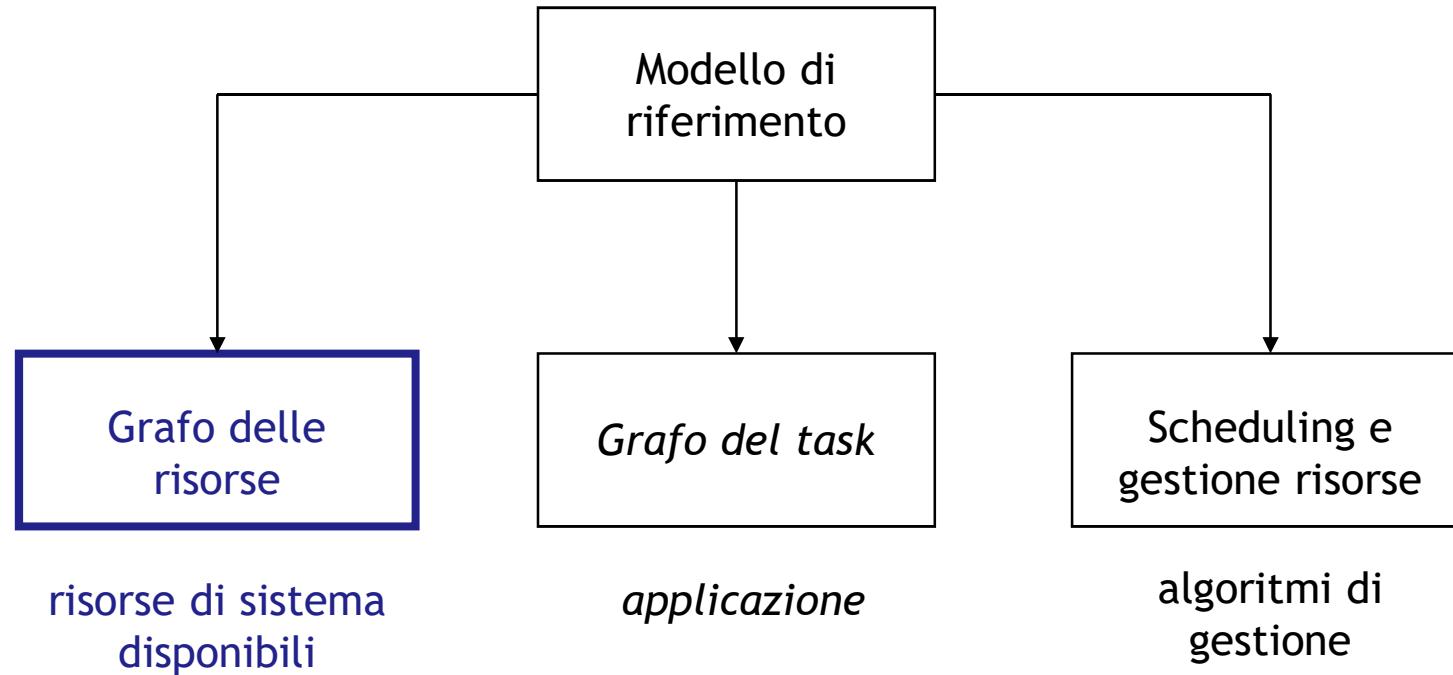
Obiettivi del modello

-
- Astrarre dalle caratteristiche funzionali dei sistemi
 - Evidenziare le proprietà temporali ed i requisiti di risorse

Struttura del modello



Modello delle risorse





Processori e risorse

- *Processori*: server, risorse attive (CPU, tritte di rete, dischi, etc.)
- *Tipi di processori*:
 - due processori sono *dello stesso tipo* se funzionalmente identici e possono essere scambiati tra loro
 - Es.: tritte di rete tra due *peer* e con lo stesso transmission rate, CPU in sistemi SMP e multicore omogenei
 - processori *di tipo diverso* non possono essere scambiati tra loro
 - Es.: differenze funzionali (CPU vs. disco) o differenze di ruolo nella topologia del sistema
- P_1, \dots, P_m



Processori e risorse

- *Risorse*: risorse passive
- Risorse necessarie, in aggiunta ai processori, per assicurare l'avanzamento della applicazione
- Non sono caratterizzate da un parametro di velocità (diversamente dai processori)
- Es.: semafori per accesso a sezione critica, lock
- Es.: data link gestito con finestra mobile
 - Job: trasmissione di un messaggio
 - Processore: data link
 - Risorsa: numero di sequenza valido
- R_1, \dots, R_s



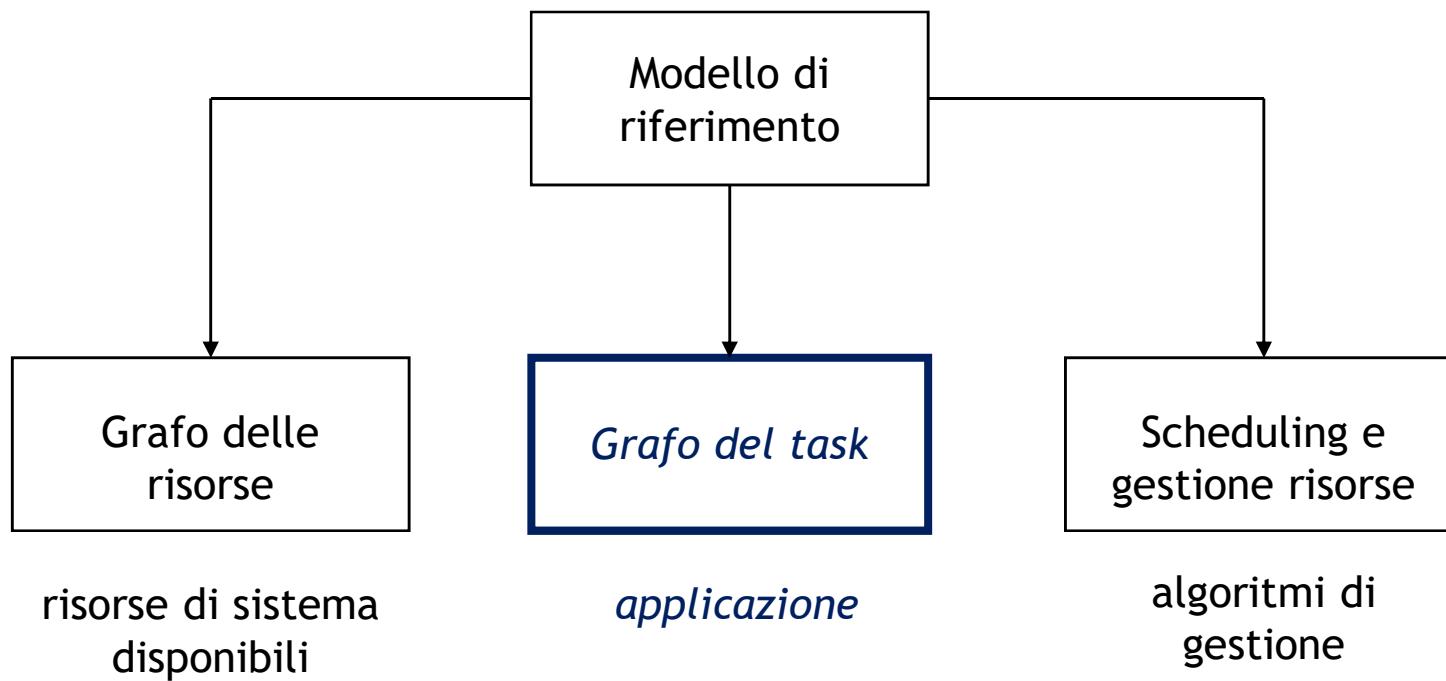
Risorse

- **Risorse *riusabili*:**
 - rese nuovamente disponibili dopo l'uso, riutilizzabili in modo sequenziale (*serially reusable*)
 - talvolta dette *serializzabili*, intendendo che devono essere assunte in modo sequenziale
 - es.: semafori, data lock, numeri di sequenza
 - possono essere divise in *tipi* ed *istanze per tipo*
- **Risorse *consumabili*:**
 - scompaiono dopo l'uso
 - es.: messaggi
- **Risorse *abbondanti*:**
 - nessun job è ritardato per l'attesa di queste risorse
 - solitamente trascurabili dai modelli
 - es.: memoria, se preallocata in modo statico e sufficiente



Modello della applicazione

- Viene detta anche carico di lavoro o *workload*





Parametri temporali

- J_i : *Job*, unità di lavoro
 - T_j o τ_j : *Task*, insieme di job correlati
 - r_i : *istante di rilascio* di J_i
 - d_i : *deadline assoluta* di J_i
 - D_i : *deadline relativa* di J_i
 - e_i o C_i : *tempo di esecuzione* (massimo) di J_i , WCET
-
- Perchè WCET?
 - la variabilità dei C_i è tipicamente ridotta *nei task RT*
 - le frazioni di tempo e risorse non utilizzate sono rese disponibili a processi *soft real-time* o *non real-time*



Modello per task periodici

- Insieme di *task* : τ_1, \dots, τ_n
- Ogni task consiste di *job* : $\tau_i = \{J_{i1}, J_{i2}, \dots\}$
- Φ_i : *fase* di τ_i , $\Phi_i = r_{i1}$ istante di rilascio del primo job
- T_i : *periodo* di τ_i , intervallo minimo tra due istanti di rilascio
- H : *iperperiodo*, $H = mcm(T_1, \dots, T_n)$
- C_i : *tempo di esecuzione* di τ_i
- U_i : *utilizzazione* di τ_i , $U_i = C_i/T_i$
- D_i : *deadline relativa* di τ_i , spesso $D_i = T_i$



Modello per task periodici

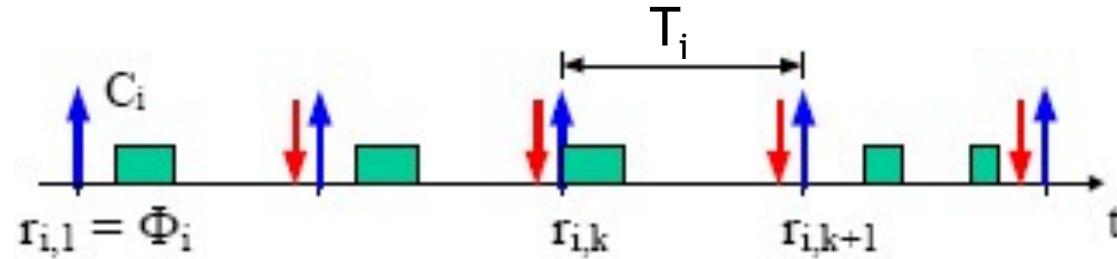
- Insieme di *task* : τ_1, \dots, τ_n
- Ogni task consiste di *job* : $\tau_i = \{J_{i1}, J_{i2}, \dots\}$
- Φ_i : *fase* di τ_i , $\Phi_i = r_{i1}$ istante di rilascio del primo job
- T_i : *periodo* di τ_i , intervallo minimo tra due istanti di rilascio
- H : *iperperiodo*, $H = mcm(T_1, \dots, T_n)$
- C_i : *tempo di esecuzione* di τ_i
- U_i : *utilizzazione* di τ_i , $U_i = C_i/T_i$
- D_i : *deadline relativa* di τ_i , spesso $D_i = T_i$

E' un caso di interesse per le applicazioni?



Modello per task periodici

- $r_{i,1} = \Phi_i$
- $r_{i,k+1} = r_{i,k} + T_i$



$$r_{i,k} = \Phi_i + (k-1)T_i$$

$$d_{i,k} = r_{i,k} + D_i$$

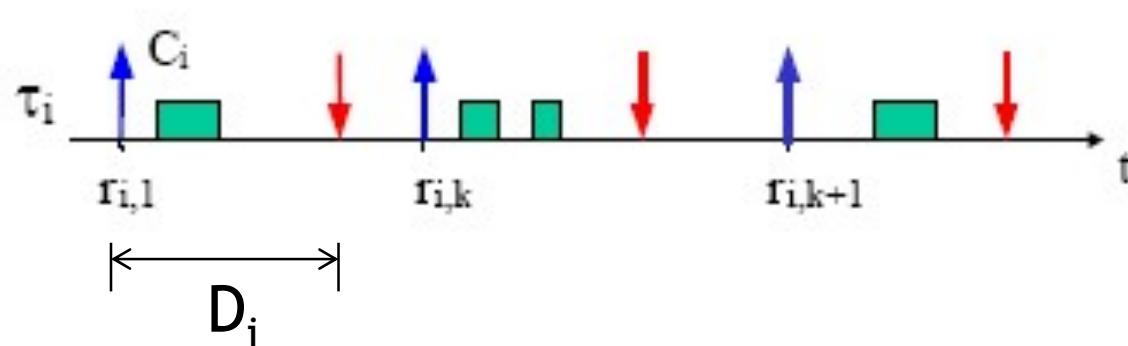
$$\text{spesso } D_i = T_i$$

τ_i caratterizzato da (C_i, T_i, D_i, Φ_i)



Modello per task aperiodici e sporadici

- Task *aperiodici*: $r_{i,k+1} > r_{i,k}$
- Task *sporadici*: $r_{i,k+1} > r_{i,k} + T_i$
- T_i è il *minimum inter-release time*





Task aperiodici e sporadici

- Spesso rappresentano eventi modellati, ma di cui non è noto l'istante di verifica
- Possono essere caratterizzati da distribuzioni dei tempi di interarrivo $A(x)$ e dei tempi di esecuzione $B(x)$
- Classificazione formale:
 - i task *aperiodici* hanno deadline *soft* o sono privi di deadline ...
 - i task *sporadici* hanno o possono avere deadline relative di tipo *hard*
- Nel mondo reale? Impossibile fornire garanzie a task di tipo hard RT altrimenti...



Task con jitter

- jitter = variabilità nei tempi di rilascio e di esecuzione
- $r_i \in \{r_i^-, r_i^+\}$ *jitter nel tempo di rilascio* (task periodici)
- $e_i \in \{e_i^-, e_i^+\}$ *jitter nel tempo di esecuzione*
- Caso peggiore: $e_i = e_i^+$, $r_i = r_i^+$
- Per un'analisi di schedulabilità si può assumere
 $r_i = r_i^-$ e $WCET = e_i^+ + (r_i^+ - r_i^-)$
- Nei task periodici talvolta il *jitter nel tempo di completamento* $\{f_i^- - r_i, f_i^+ - r_i\}$ è un problema in sè e va minimizzato



Vincoli di precedenza

- Rappresentati con un *grafo di precedenza*
- Esprimono *dipendenze* tra dati e di controllo

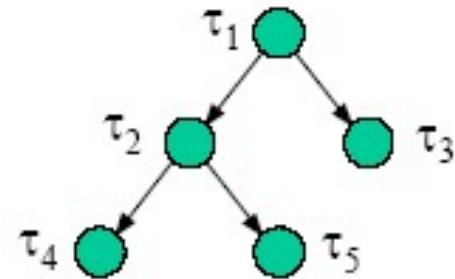
- *Relazione di precedenza*: $<$ (ordinamento parziale)
- *Grafo di precedenza*: $G=(J, <)$
- Esempi di vincoli di precedenza: vincoli AND/OR

- Non tutti i vincoli di precedenza sono rappresentabili in un grafo di precedenza tra task (ad es. accesso esclusivo a dati condivisi)
- Esistono strumenti formali in grado di esprimere precedenze e sincronizzazioni → *Reti di Petri*



Grafo di precedenza

- Grafo orientato aciclico (DAG - Direct Acyclic Graph)



τ_1 predecessore di τ_4 :

$$\tau_1 < \tau_4$$

τ_1 predecessore immediato di τ_2 :

$$\tau_1 \rightarrow \tau_2$$



Parametri funzionali

□ Revocabilità

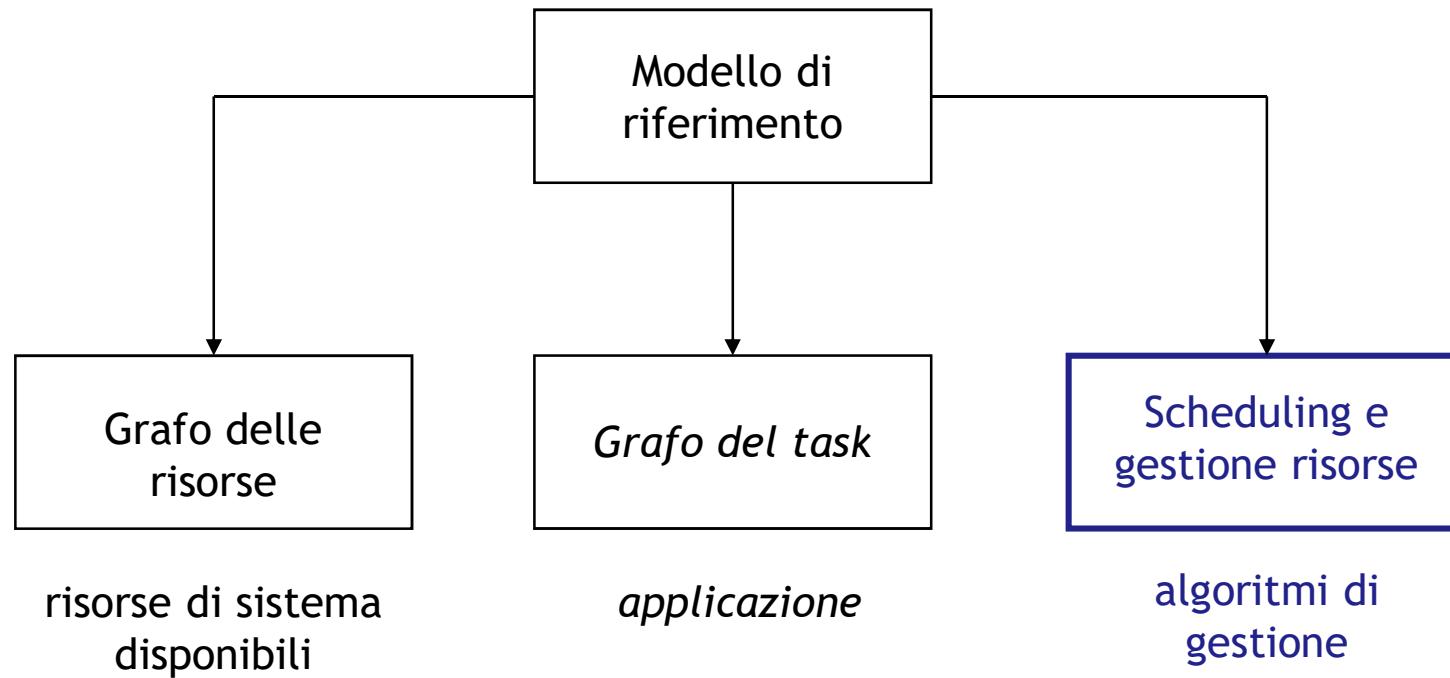
- *revoca* o *preemption*: sospensione dell'esecuzione di un job per cedere il processore ad un job più urgente
- la non-revocabilità è spesso legata ad una specifica risorsa; il job può essere ancora revocabile su altre risorse
- la *preemption* ha un costo

□ Criticità

- possiamo associare un peso o valore ai job per indicarne la criticità relativa
- schedulatori e protocolli di accesso alle risorse possono ottimizzare misure di prestazioni che tengano conto di tali pesi



Modello dell'algoritmo di gestione





Schedule ed algoritmi di scheduling

- *schedule*: assegnamento di job ai processori disponibili
- *schedule fattibile (feasible)*: nella schedule ogni job inizia l'esecuzione *non prima* dell'istante di rilascio e completa *entro* la sua deadline
- *ottimalità*: un algoritmo di scheduling è ottimo se è in grado di produrre sempre una schedule fattibile quando essa esiste
- *misure di prestazione*:
 - numero di job in ritardo (tardy jobs)
 - tardiness massima o media
 - tempo di risposta massimo o medio
 - makespan



Se schedule non fattibile?

- Valutare possibilità e fattibilità di una schedule con processore più veloce
 - Valutare partizionamento dei task e assegnazione a core multipli
 - Quanti e quali task risultano garantiti dall'algoritmo in esame?
 - Modificare le caratteristiche dei task, se consentito dall'applicazione
 - Nel caso di task periodici, valutare se è possibile intervenire sui parametri di uno o più task: T_i ? C_i ? D_i ?
 - Ridiscutere le specifiche con il committente
 - Understand and optimize, before giving up!
-



Misure di prestazione per job soft RT

- La metrica più utilizzata è il *tempo di risposta medio*
- Nei sistemi RT hard/soft misti, l'obiettivo tipico è garantire il rispetto delle deadline dei job hard minimizzando il tempo di risposta medio dei job soft
- Non c'è vantaggio a completare in anticipo i job hard,
→ è possibile ritardarne l'esecuzione per migliorare la risposta ai job soft
- Altre metriche:
 - *miss rate*: percentuale di job completati in ritardo
 - *loss rate*: percentuale di job non eseguiti (ad es. scartati)
 - *invalid rate*: miss rate + loss rate



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Problemi e tipologie di scheduling in tempo reale

prof. Stefano Caselli
stefano.caselli@unipr.it



Problemi nella schedulazione real-time

- Alcuni esempi che testimoniano che elaborazione *in tempo reale* non equivale ad elaborazione *veloce*
- Situazioni in cui modifiche ad un problema che dovrebbero facilitarne la soluzione producono invece tempi di risposta maggiori → *anomalie*

- Analisi di sequenze di *esecuzione in sistemi multiprocessore* e problemi correlati



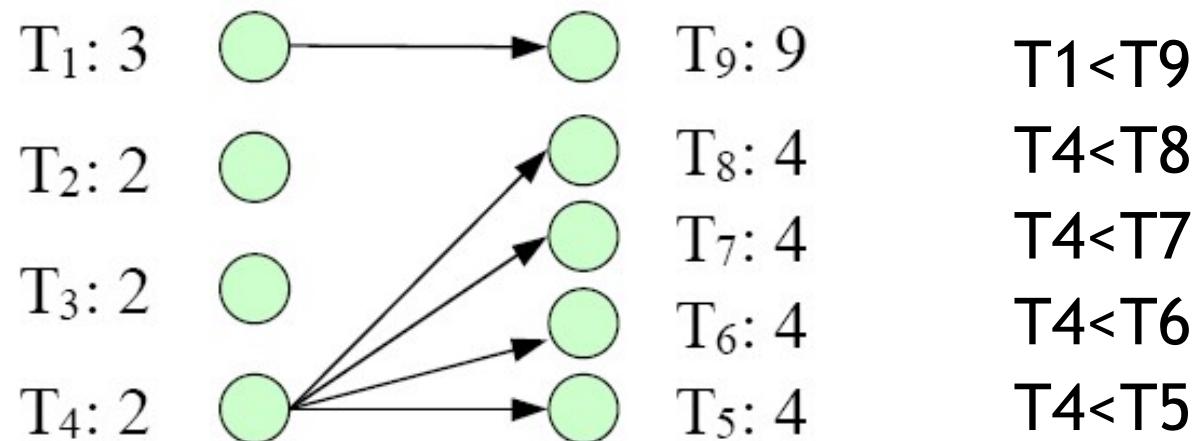
Scenario

- Insieme di task con vincoli di precedenza, organizzati a grafo
- In questo caso: task aperiodici in esecuzione *one-shot*
- Per un insieme di task aperiodici *correlati*, una metrica spesso utilizzata di valutazione complessiva è il tempo di completamento dell'insieme di task, ovvero il *makespan*
 - Il makespan è il tempo che intercorre dal rilascio del primo task al completamento dell'ultimo
- Confronteremo il makespan con un valore *deadline*



Esempio

- Insieme di task: $T = \{T_1, \dots, T_9\}$
- Priorità: $\text{Pri}(T_i) > \text{Pri}(T_j)$ per $\forall i < j$
- Vincoli di precedenza e tempi di esecuzione:

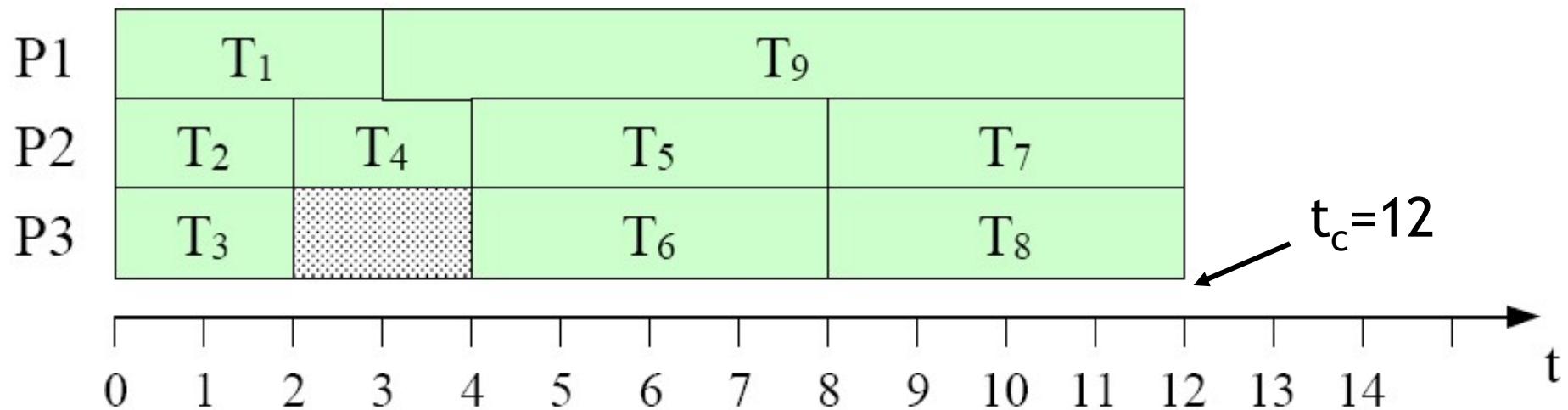
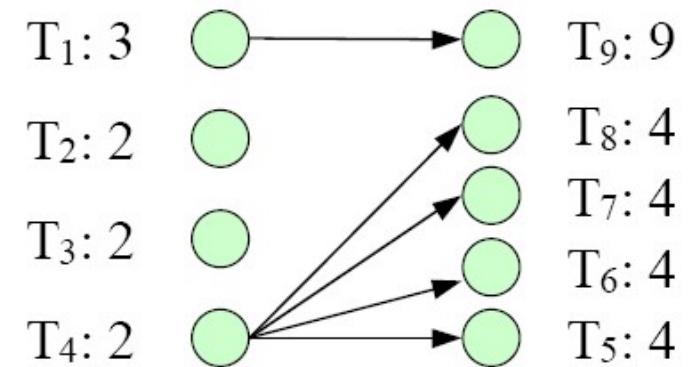


- Sistema di elaborazione: k processori P_1, \dots, P_k



Anomalie di scheduling

- Esecuzione su $k=3$ processori:





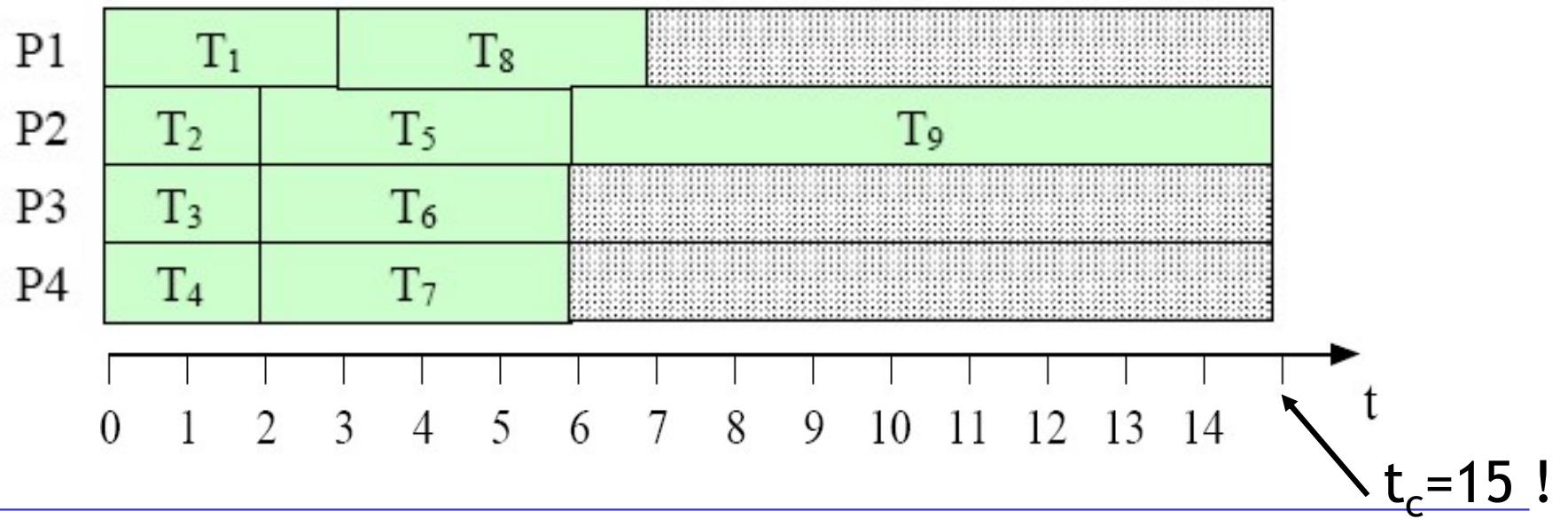
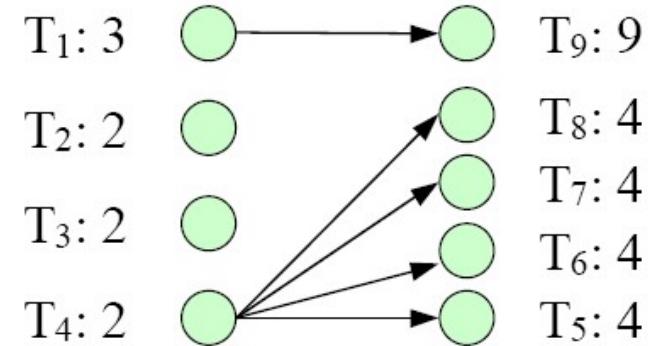
[Esercizio]

- Costruire la schedule per le stesse condizioni del caso precedente nell'ipotesi in cui $\text{Pri}(T_i) < \text{Pri}(T_j)$ per $\forall i < j$
 - Quanto vale il *makespan*?
-
- La *priorità* attribuita influenza le prestazioni!
 - È un parametro libero o vincolato?



Anomalie di scheduling

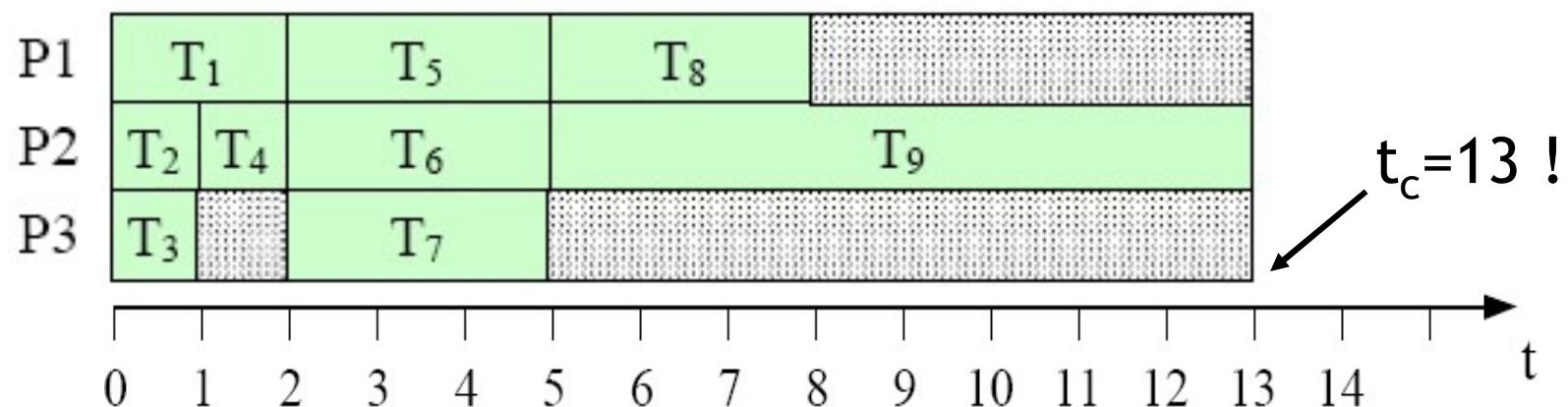
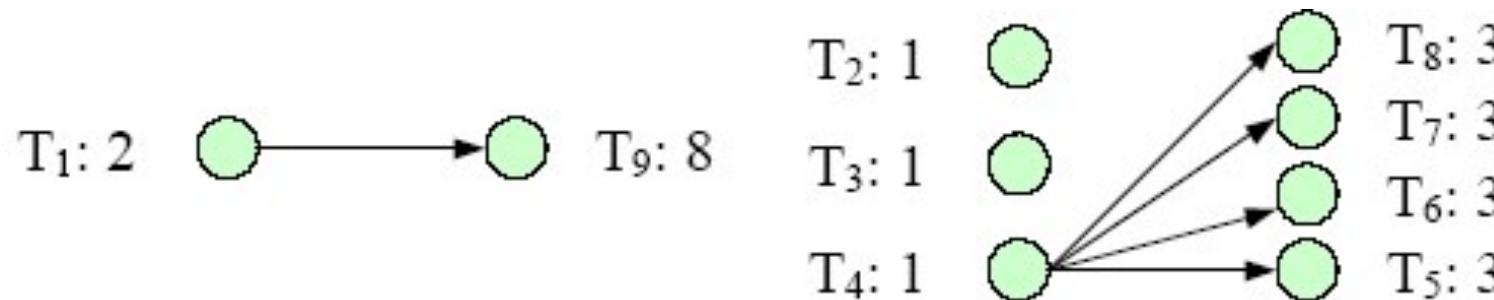
- Esecuzione su $K=4$ processori:





Anomalie di scheduling

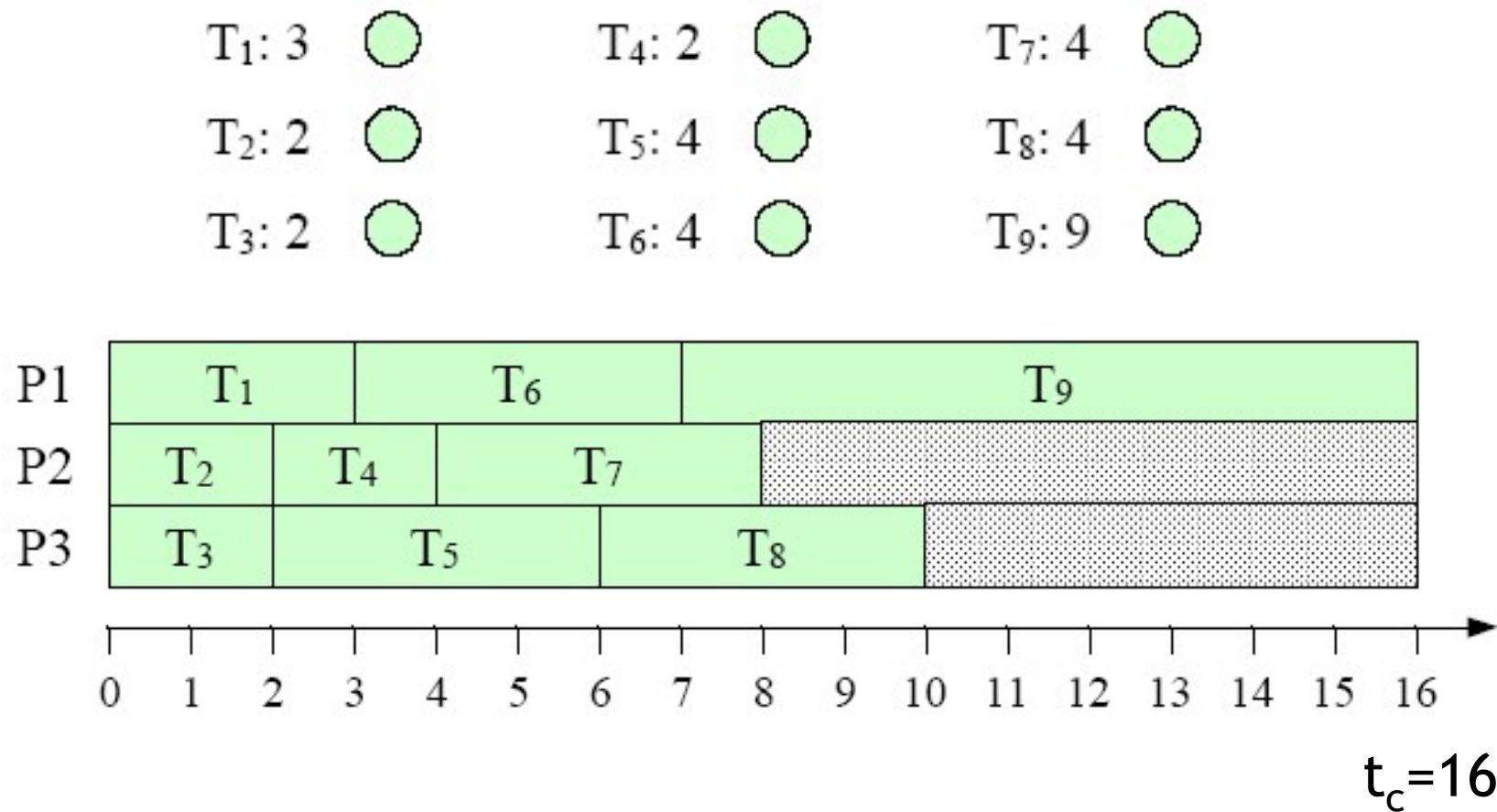
- Esecuzione di task più brevi, $C_i \rightarrow C_i - 1 \forall i$ (ad es. per ottimizzazione del codice):





Anomalie di scheduling

- Rimozione dei *vincoli di precedenza*:

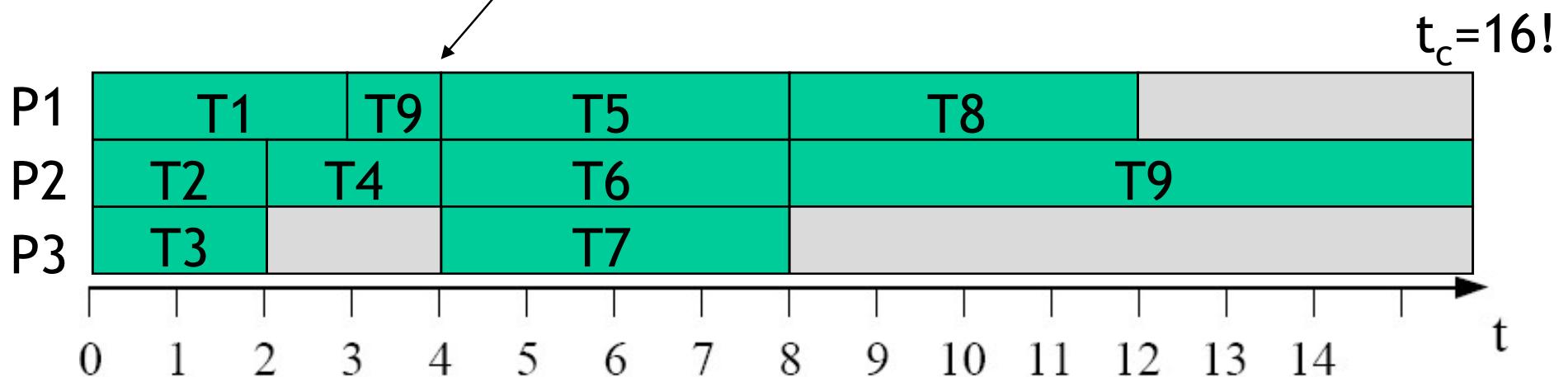




Anomalie di scheduling

□ Introduzione di *preemption*:

preemption





Anomalie di scheduling

- *Anomalie di Richard*
- Teorema (Graham, 1969):

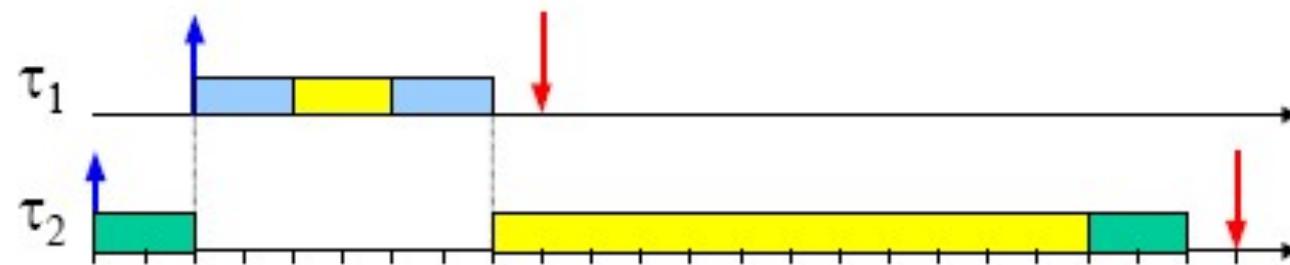
Per un task set schedulato in modo ottimale su un sistema multiprocessore con assegnamento di priorità, numero di processori, tempi di esecuzione e vincoli di precedenza fissati, l'aumento del numero di processori, la riduzione dei tempi di esecuzione ed il rilassamento dei vincoli di precedenza possono aumentare la lunghezza della schedule.

- → modificando anche lievemente il problema in modo da rendere *più semplice* il rispetto delle scadenze, il tempo di risposta può divenire più elevato

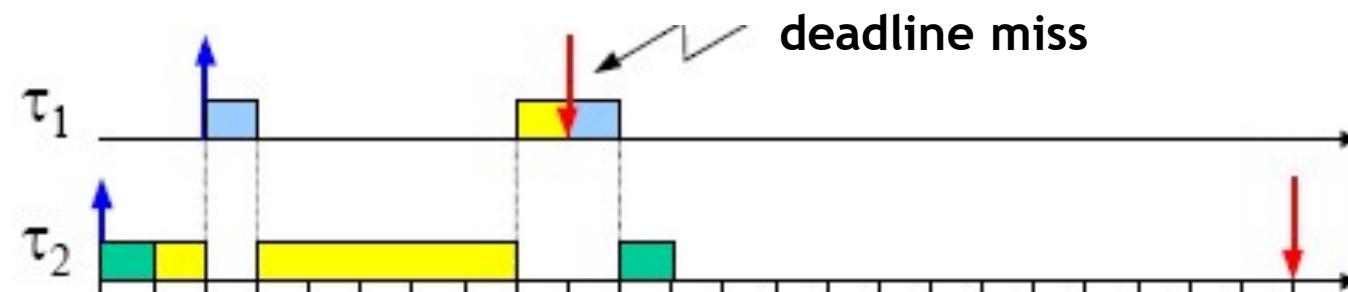


Anomalie di scheduling in presenza di vincoli su risorse

- Presenza di sezioni critiche:



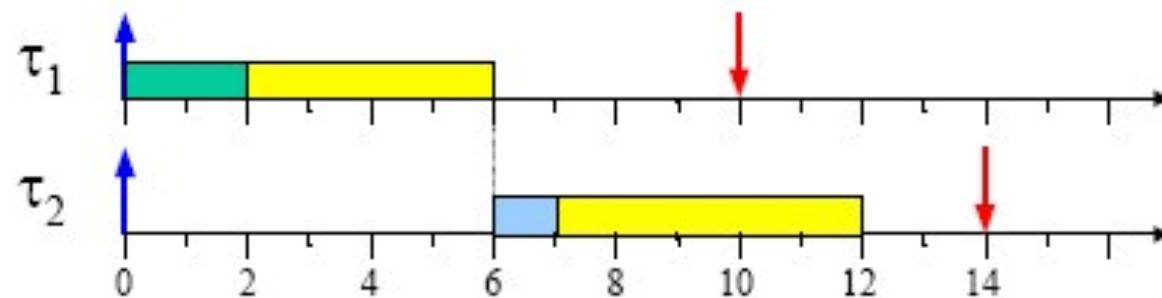
- Con velocità processore 2x:





Anomalie di scheduling in presenza di vincoli su risorse

- Introduzione di un ritardo D:



- Un ritardo D (es. sleep D) prima della sezione critica da parte di τ_1 può determinare un ritardo superiore a quello previsto o una deadline miss



Problema di scheduling generale

- Il problema di assegnare processori e risorse ai job soddisfacendo vincoli di precedenza e temporali è, nel caso generale, NP-completo e quindi *intrattabile*
- Nei sistemi real-time dinamici, le decisioni di scheduling devono essere prese *on-line*, durante l'esecuzione dei task
- Possibili elementi di semplificazione:
sistemi monoprocessoressi, assenza di vincoli di precedenza o di risorse, preemption, priorità statiche, task omogenei, rilascio simultaneo dei task



Classificazione degli algoritmi di scheduling

- *preemptive/non-preemptive*
 - *preemptive*: il task in esecuzione può essere interrotto per assegnare il processore ad un altro task;
 - *fully preemptive*: il task può essere interrotto in qualsiasi istante
 - *non-preemptive*: il task esegue fino al completamento; tutte le decisioni di scheduling sono prese quando il task completa l'esecuzione
- *statici/dinamici*
 - *statici*: le decisioni sono prese in base a parametri fissi, attribuiti ai task prima della loro attivazione
 - *dinamici*: le decisioni sono prese in base a parametri dinamici, variabili a tempo di esecuzione



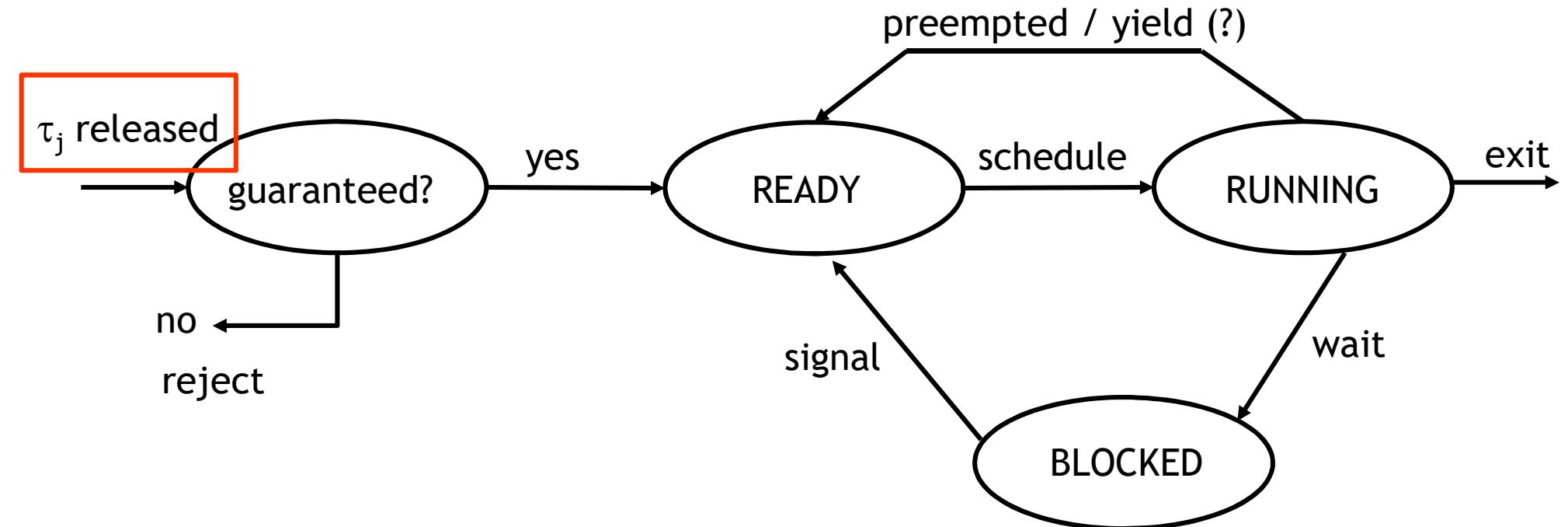
Classificazione degli algoritmi di scheduling

- *on-line/off-line*
 - *on-line*: le decisioni di scheduling sono prese on-line quando un nuovo task entra nel sistema o un task in esecuzione termina
 - *off-line*: l'algoritmo è eseguito sull'intero task set prima dell'esecuzione; la schedule è memorizzata in tabella ed utilizzata dal dispatcher
- *garantiti/best-effort*
 - *garantiti*: i task sono garantiti e rispettano le proprie deadline; nuovi task vengono *accettati* solo se l'intero task set resta garantito
 - *best-effort*: le deadline sono rispettate quando possibile; nuovi task vengono accettati indipendentemente dal loro impatto sulla schedule; in *media* forniscono, sul task set, risultati migliori rispetto a quelli garantiti



Sistemi con garanzia

- Per poter garantire task real time a tempo di esecuzione occorre un *sistema di accettazione*





Classificazione degli algoritmi di scheduling

- *ottimi/euristici*
 - *ottimi*: garantiscono di trovare una schedule fattibile se esiste; in presenza di una metrica di valore, trovano la schedule con il valore massimo
 - *euristici*: tendono a fornire buoni risultati senza garanzie di ottimalità della schedule

- *chiaroveggenti*
 - conoscono in anticipo gli istanti di arrivo dei task



Schedulabile?

- *Ripasso:*
- *schedule*: assegnamento di job ai processori disponibili
- *schedule fattibile*: nella schedule ogni job inizia l'esecuzione non prima dell'istante di rilascio e completa entro la sua deadline
- *ottimalità*: un algoritmo di scheduling è ottimo se è in grado di produrre sempre una schedule fattibile quando essa esiste

- → *insieme di task schedulabile*: insieme di task per il quale esiste una schedule fattibile



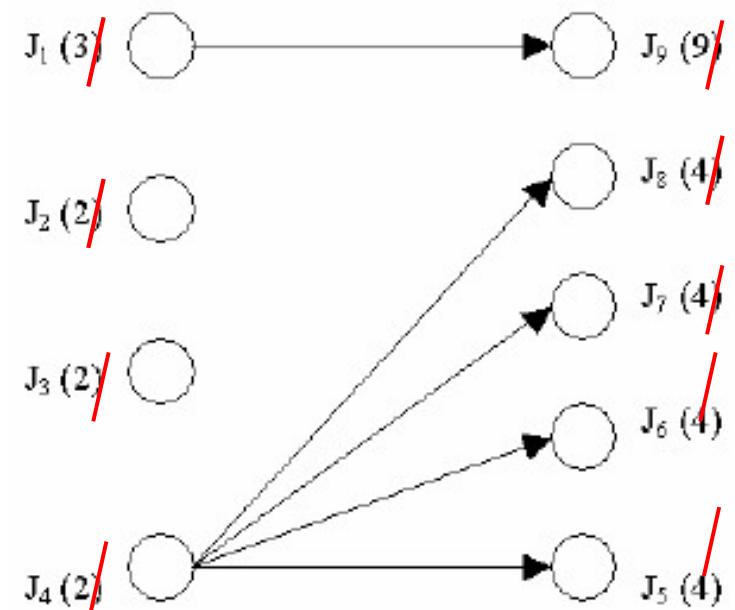
Schedulabile da Γ ?

- Un *insieme di task* Ω è *schedulabile dall'algoritmo* Γ se Γ produce una schedule fattibile per i task in Ω
 - L'algoritmo Γ potrebbe comunque non essere ottimo
- Il quesito sulla *schedulabilità* di un *insieme di task* Ω da parte di un algoritmo Γ in base ad un assegnato criterio di analisi ξ ammette in generale le risposte **sì | no | forse**
- Il quesito sulla *garanzia* di un *job o task* $\tau_i \in \Omega$ da parte di un algoritmo Γ sulla base di un criterio di analisi ξ ammette solo le risposte **sì | no**



Esercizio 1

- Calcolare makespan con esecuzione di task *più brevi*, $C_i \rightarrow C_i - 1 \forall i$ e $K=4$ processori
 - NB: correggere le durate rispetto a quelle iniziali, riportate in figura!





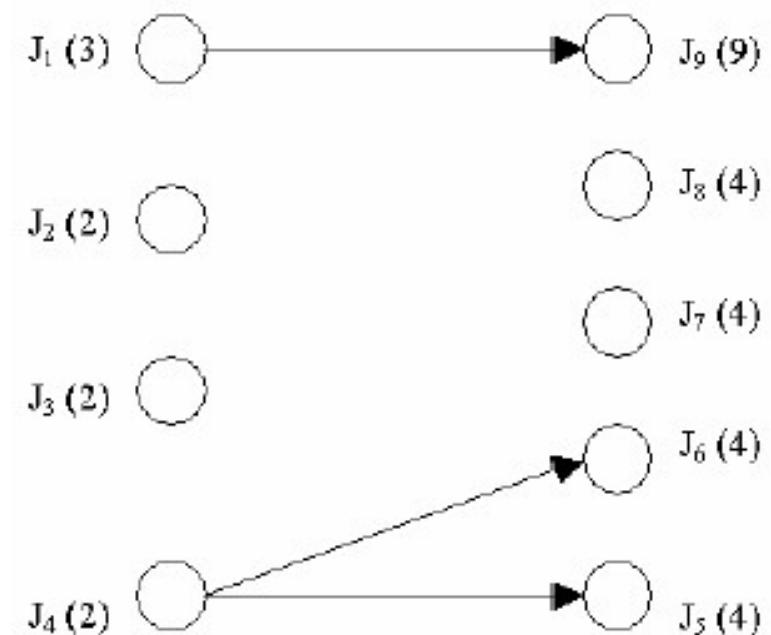
Esercizio 2

- Calcolare il makespan con 3 processori e *riduzione* dei vincoli di precedenza:

$J_1 < J_9$

$J_4 < J_6$

$J_4 < J_5$





Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Approcci allo scheduling real-time

prof. Stefano Caselli

stefano.caselli@unipr.it

<http://rimlab.ce.unipr.it>



Approcci principali allo scheduling real-time

- Schedulatori di tipo *clock-driven* (o *time-driven*):
Le decisioni di scheduling sono prese *in specifici istanti di tempo*, tipicamente scelti *a priori*

- Schedulatori di tipo *priority-driven*:
Le decisioni di scheduling sono prese *quando si verificano particolari eventi* nel sistema, ad es.:
 - un job diventa pronto
 - il processore diventa libero



Schedulatori work-conserving

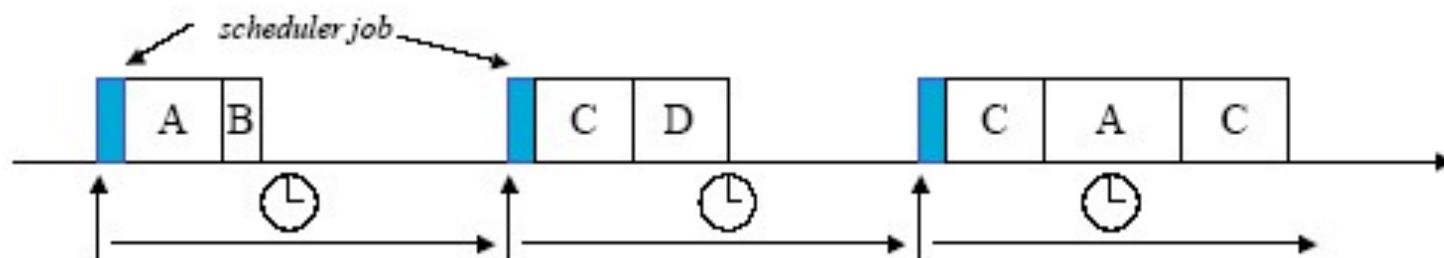
- Scheduling *work-conserving*:
il processore è comunque *impegnato* nell'esecuzione di un job
se ci sono job da completare

- Scheduling *non work-conserving*:
in qualche situazione il processore può essere mantenuto
libero anche in presenza di job da completare



Approccio *clock-driven*

- *Istante di decisione di scheduling*: istante in cui lo scheduler decide quale job eseguire successivamente
- Negli scheduler clock-driven, gli istanti di decisione sono definiti *a priori*
- Ad esempio: lo scheduler viene risvegliato periodicamente e genera una porzione di schedule





Approccio *clock-driven*

- Quando i parametri dei job sono noti a priori, la schedule può essere pre-calcolata fuori linea e memorizzata in una tabella
→ schedulatori *table-driven*
- In alternativa: negli istanti predefiniti di decisione lo scheduler utilizza la priorità per decidere l'ordine dei job da mettere in esecuzione; lo scheduler resta *clock-driven*!
- La pianificazione statica della schedule o degli istanti di decisione può mantenere il processore *idle* anche in presenza di job pronti



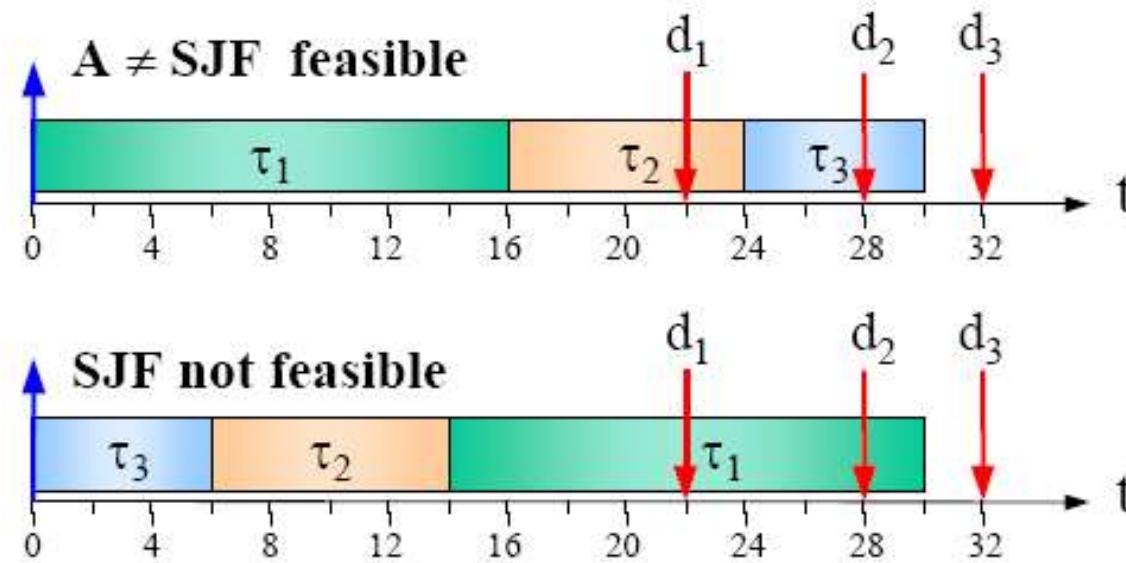
Approccio *priority-driven*

- Criterio base: il processore non resta mai inattivo se c'è del lavoro da svolgere (schedulatori *work-conserving*)
- Il job viene scelto da una lista gestita con un *criterio di priorità*. Esempi: priorità statica, FIFO, LIFO, SET, LET, EDF
- La priorità attribuita a livello utente in generale non coincide con la priorità attribuita al job dal SO e su cui il SO basa le proprie decisioni di scheduling
- La priorità può influenzare le decisioni di scheduling in modo *preemptive* o *non-preemptive*



Approccio *priority-driven*

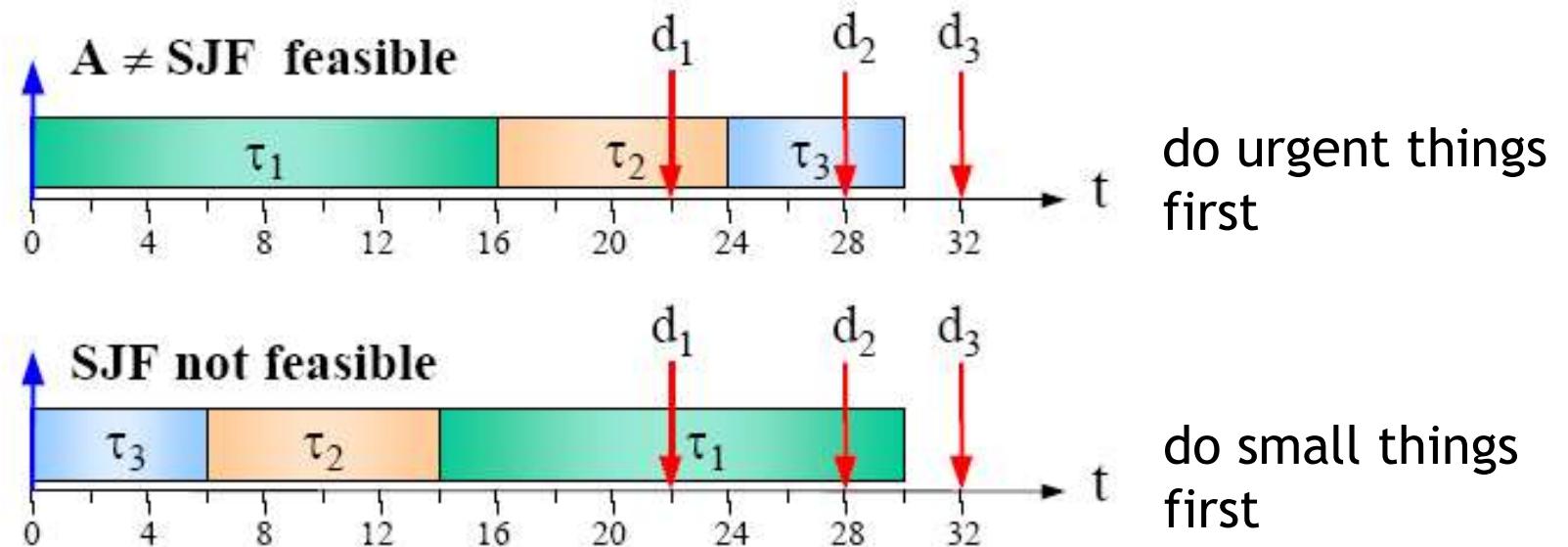
- Esempi di algoritmi di ordinamento delle priorità che considerano le caratteristiche dei job: LET, SET, EDF, LRT, LSF
- SJF (ovvero SET) è ottimo nello scheduling general-purpose ma non lo è nei sistemi RT perchè ignora le deadline:





Approccio priority-driven

- Esempi di algoritmi di ordinamento delle priorità che considerano le caratteristiche dei job: LET, SET, EDF, LRT, LSF
- SJF (ovvero SET) è ottimo nello scheduling general-purpose ma non lo è nei sistemi RT perché ignora le deadline:





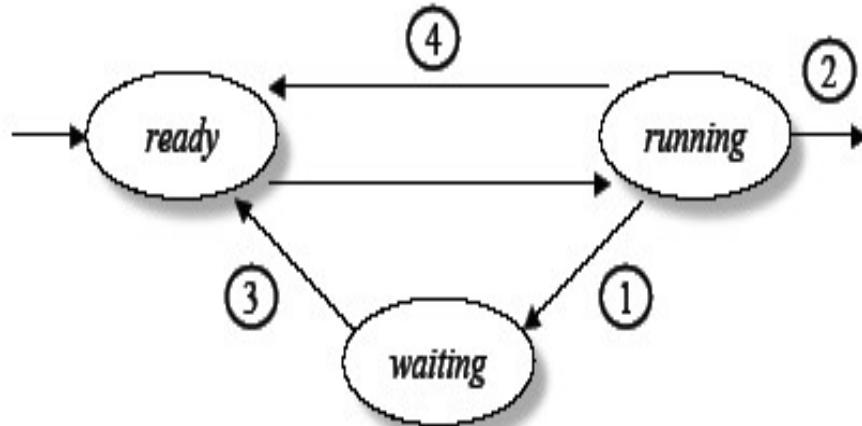
Approccio *priority-driven*

- Una possibile realizzazione di uno scheduling priority-driven di tipo *preemptive*:
 - Assegna le priorità *ai job*
 - Prendi le decisioni di scheduling quando:
 - un job diventa pronto
 - il processore diventa libero
 - le priorità dei job cambiano
 - In ciascun istante di decisione di scheduling, scegli il job a priorità più elevata
- La priorità dei job può essere statica o dinamica (es. slack time scheduling)



Approccio priority-driven

- Istanti di decisione di scheduling:



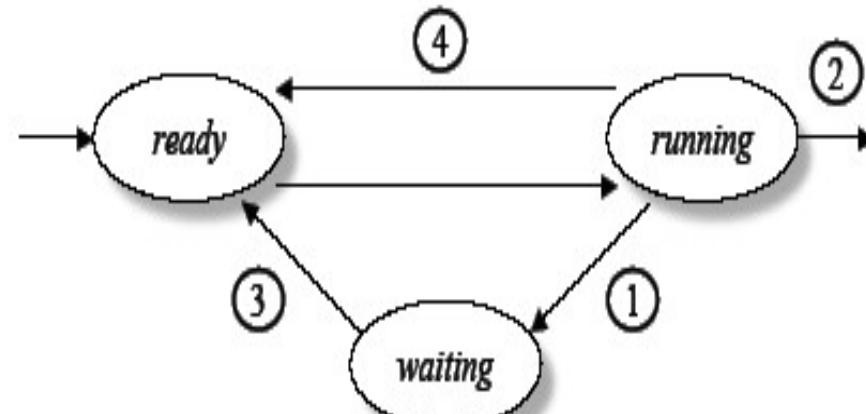
- il job passa da esecuzione a waiting
- il job in esecuzione termina
- un job in attesa diviene pronto
- il job in esecuzione torna nello stato pronto

- In genere le priorità dei job possono cambiare solo in corrispondenza alle transizioni di stato



Approccio priority-driven

- In uno scheduler priority-driven *non-preemptive*, le decisioni di scheduling sono prese solo quando:
 - il processore diventa libero (1), (2)
 - un job diventa pronto e il processore è libero (3 con processore libero)
 - Il job in esecuzione esegue una yield (4)



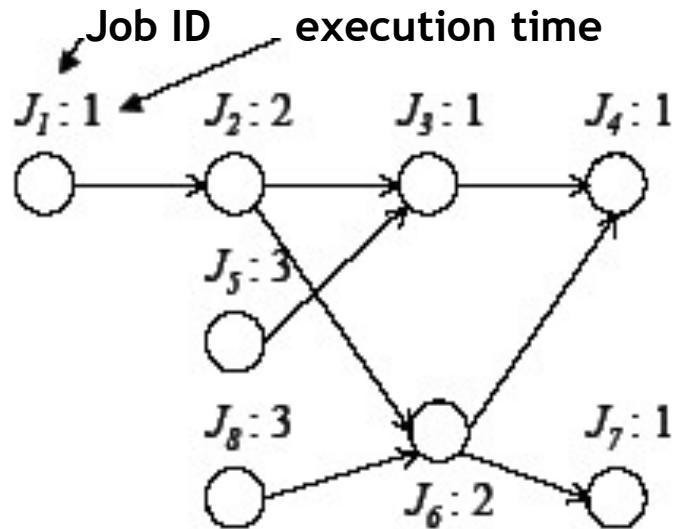


Elementi aggiuntivi del problema di scheduling

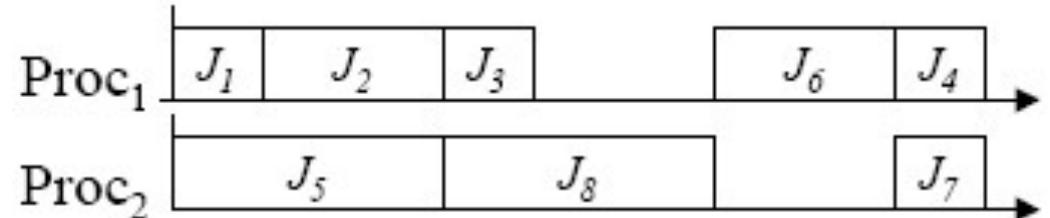
- Esistenza di vincoli di precedenza tra job
- Coerenza tra vincoli di precedenza e vincoli temporali
- Presenza o assenza di preemption
- Criterio di priorità in presenza di vincoli temporali, vincoli di precedenza, preemption o non preemption



Schedule basate su priorità in assenza di revoca



- Grafo con vincoli di precedenza
- Vincoli temporali: $r_i=0$, $d_i=t^* \quad \forall i$
- Schedule su 2 processori, con $\text{Pri}(J_i) > \text{Pri}(J_j)$ per $\forall i < j$:

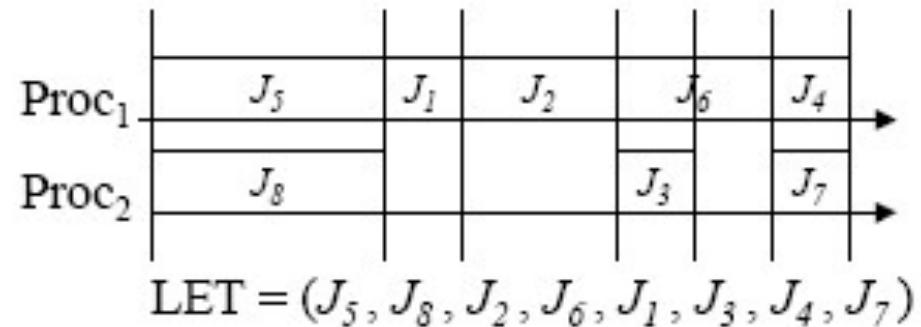
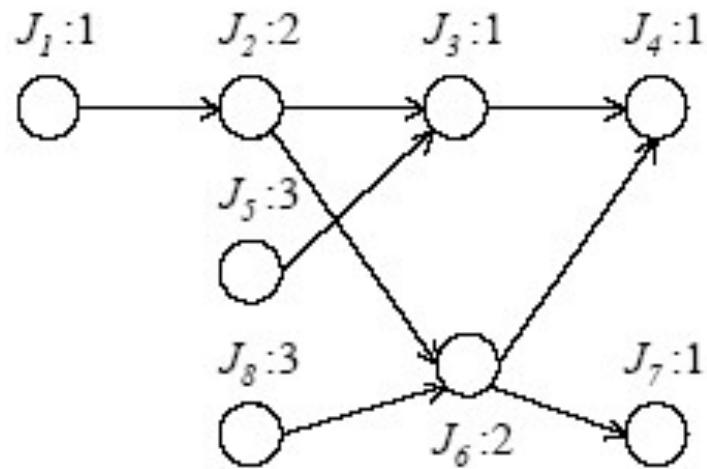


$t_c=9$



Schedule basate su priorità in assenza di revoca

- Schedule con algoritmo LET:

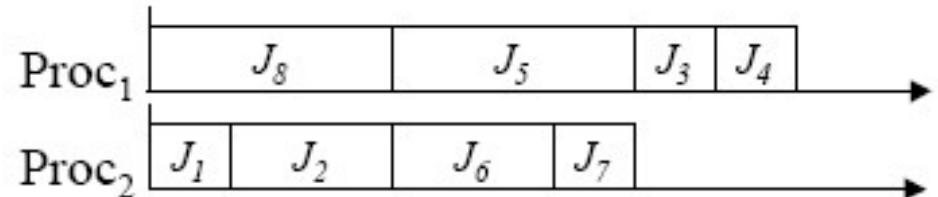
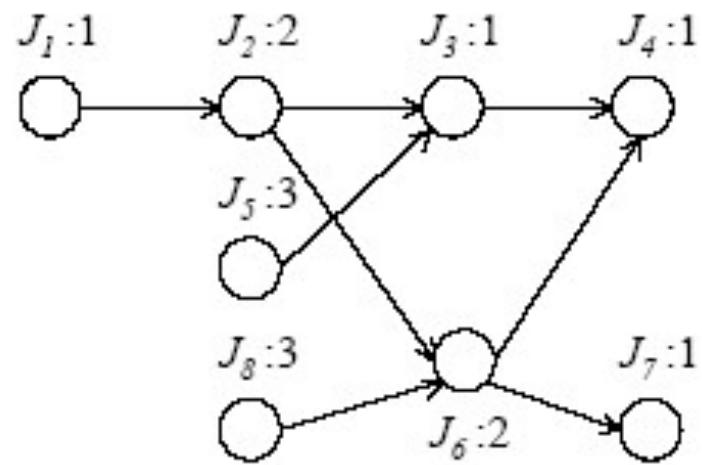


$$t_c = 9$$



Schedule basate su priorità in assenza di revoca

- Schedule alternativa:

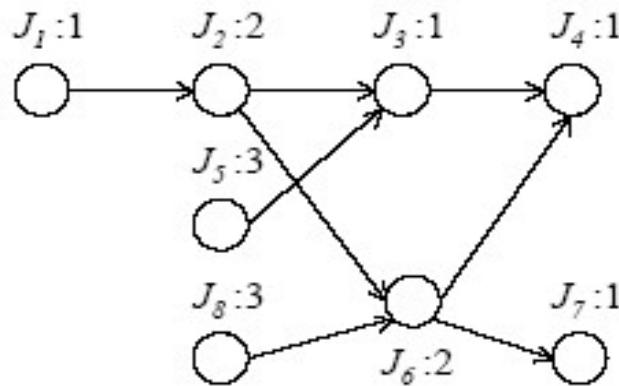


$$L = (J_8, J_1, J_2, J_3, J_4, J_5, J_6, J_7)$$

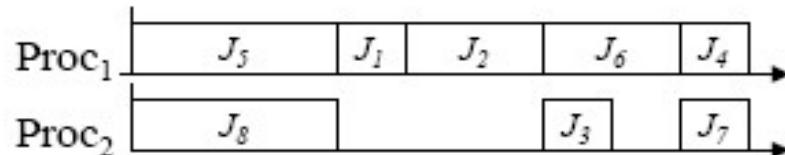
$$t_c = 8$$



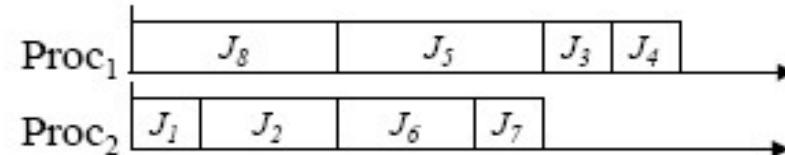
Schedule basate su priorità in assenza di revoca



$$L = (J_1, J_2, J_3, J_4, J_5, J_6, J_7, J_8)$$



$$LET = (J_5, J_8, J_2, J_6, J_1, J_3, J_4, J_7)$$



$$L = (J_8, J_1, J_2, J_3, J_4, J_5, J_6, J_7)$$

- la priorità può essere scelta arbitrariamente ?
- determina schedule e tempi di completamento diversi



Vincoli temporali effettivi

- I vincoli temporali possono essere *non consistenti* con i vincoli di precedenza
 - Esempio: $d_1 > d_2$ mentre $J_1 \rightarrow J_2$
- Per il caso *uniprocessore* è possibile rimuovere i vincoli di precedenza considerando i *vincoli temporali effettivi*
- *Istante di rilascio effettivo*:
$$r_i^{\text{eff}} = \max \{ r_i, \{ r_j^{\text{eff}} \mid J_j \rightarrow J_i \} \}$$
- *Deadline effettiva*:
$$d_i^{\text{eff}} = \min \{ d_i, \{ d_j^{\text{eff}} \mid J_i \rightarrow J_j \} \}$$



Vincoli temporali effettivi

- Vincoli temporali effettivi = vincoli temporali consistenti con i vincoli di precedenza
- Job con predecessori: → istante di rilascio effettivo
- Job con successori: → deadline effettiva
- Il calcolo dei vincoli temporali effettivi ha costo $O(n^2)$

- Teorema
Un insieme di job J è *schedulabile* su un processore se e solo se può essere schedulato in modo da rispettare i tempi di rilascio e le deadline *effettivi* dei job.



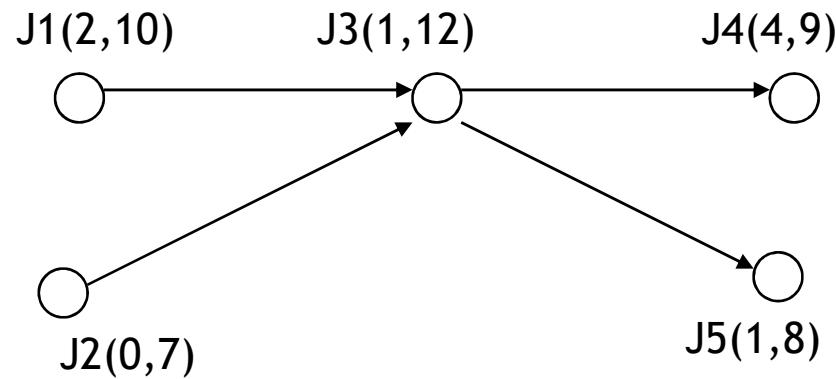
Vincoli temporali effettivi

- Il calcolo dei vincoli temporali effettivi prescinde dai tempi di esecuzione
- Il teorema tuttavia ci dice che esiste una schedule fattibile
- E' possibile che una schedule generata a partire dai vincoli temporali effettivi sia non corretta; tuttavia in tal caso è certamente possibile scambiare l'ordine di qualche job per produrne una corretta



Vincoli temporali effettivi

- Esempio - DAG con $J_i(r_i, d_i)$:



- Dopo il calcolo dei vincoli effettivi:

$J1(2,8), J2(0,7), J3(2,8), J4(4,9), J5(2,8)$



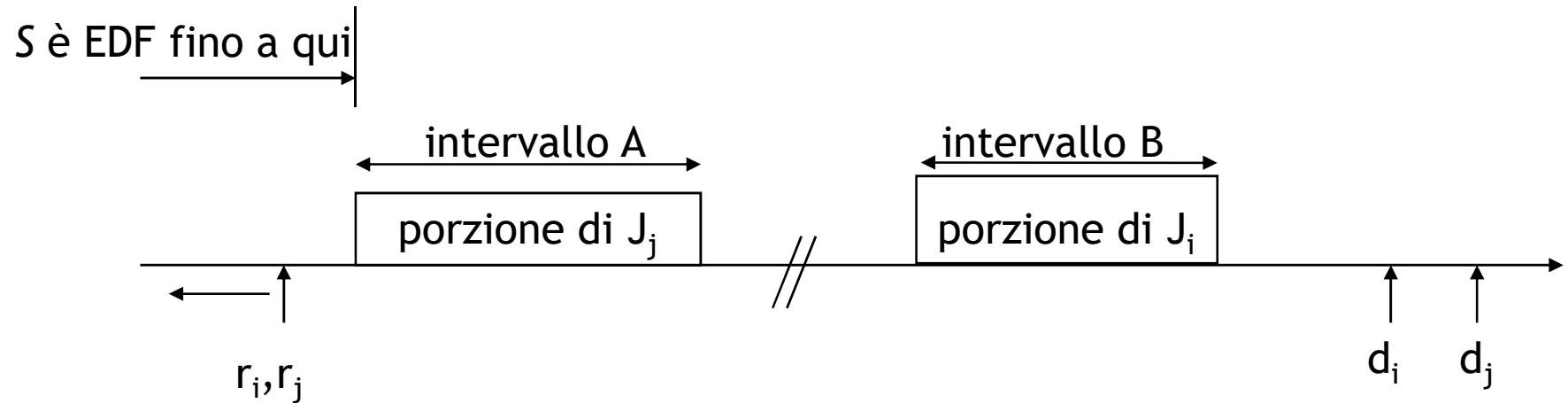
L'algoritmo EDF (Horn, 1974)

- **Algoritmo EDF (Earliest Deadline First):**
In ciascun istante, esegui il job *disponibile* (pronto o già in esecuzione) con la *deadline* più prossima
- **Teorema (Ottimalità di EDF) (Dertouzos, 1974):**
In un sistema *monoprocessore con preemption*, EDF può generare una schedule fattibile per un insieme di job J con istanti di rilascio e deadline arbitrari se e solo se tale schedule esiste
- **Dimostrazione:** mediante trasformazione della schedule



Ottimalità di EDF

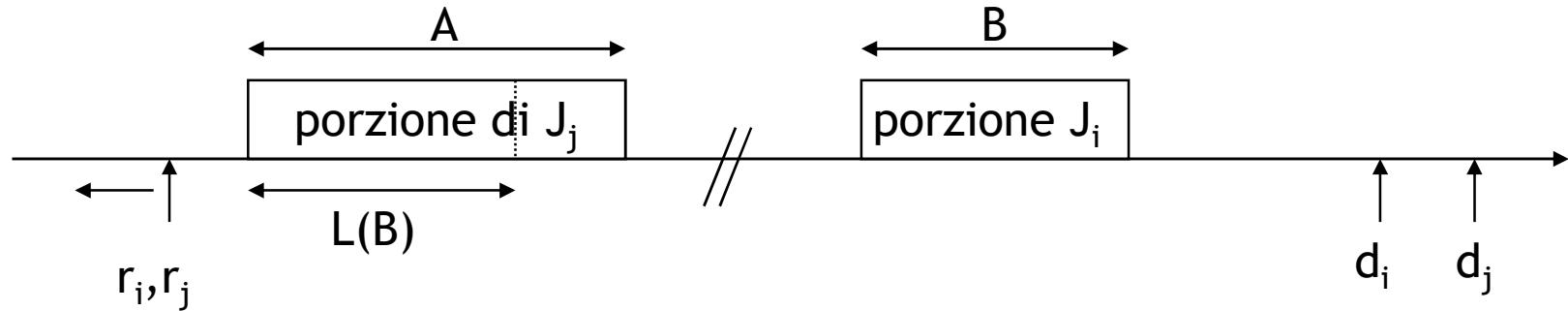
- Si ipotizzi che un'arbitraria schedule S rispetti i vincoli temporali, mentre la schedule generata da EDF non rispetti i vincoli (ovvero: EDF non è ottimo)
- Se S non è una schedule EDF, si deve verificare la seguente situazione:





Ottimalità di EDF (2)

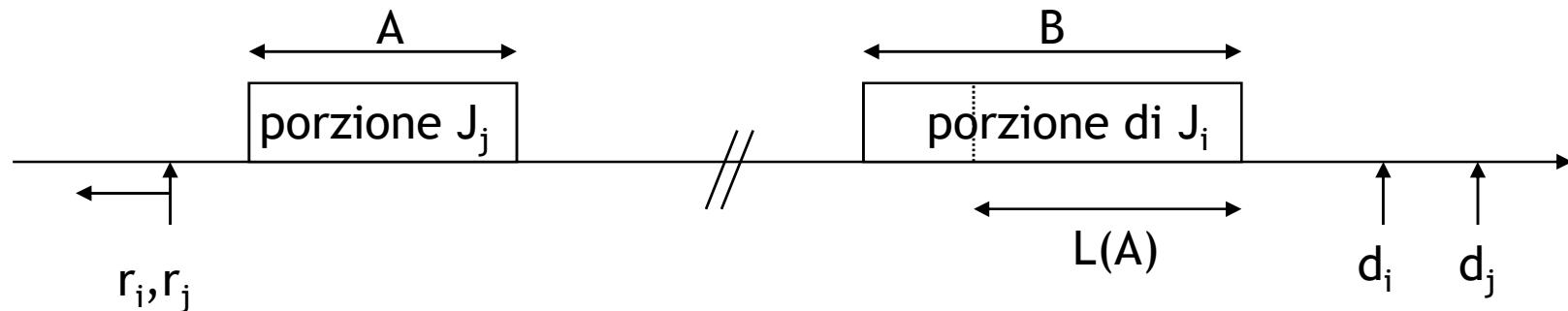
- Si possono avere due casi, e per entrambi è possibile una trasformazione che mantiene la fattibilità della schedule
- Caso 1: $L(A) > L(B)$
 - E' possibile anticipare l'esecuzione di J_i in A; J_j inizia in A e completa in B
 - Possibile perché J_j è revocabile





Ottimalità di EDF (3)

- Caso 2: $L(A) \leq L(B)$
 - J_i inizia l'esecuzione in A e la completa nella prima parte di B
 - Possibile perché J_i è revocabile



- In entrambi i casi la schedule risulta riordinata in modo EDF ed è fattibile



Ottimalità di EDF (4)

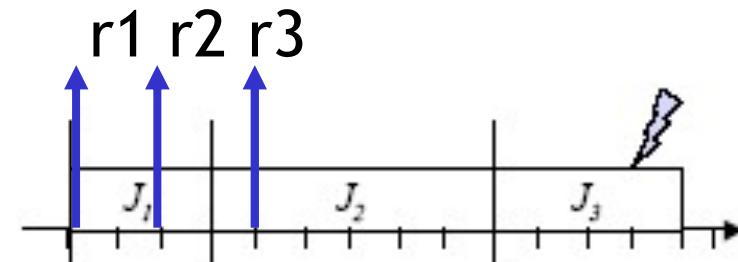
- Si ripete la trasformazione della schedule per tutte le coppie di job non scheduled in modo EDF
- Si eliminano eventuali intervalli *idle* (scheduling non work-conserving) anticipando l'esecuzione dei job pronti
- La schedule risultante è EDF...
- Il teorema segue per contraddizione □



EDF è ottimo solo nelle ipotesi date...

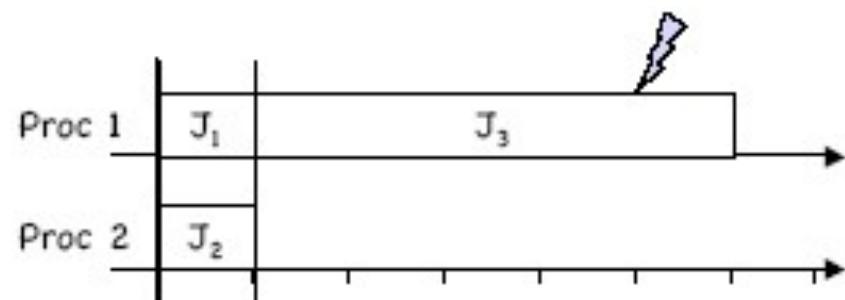
- Caso 1: Preemption non consentita:

	r_i	d_i	e_i
J_1	=	(0, 10, 3)	
J_2	=	(2, 14, 6)	
J_3	=	(4, 12, 4)	



- Caso 2: Sistema multiprocessore:

	r_i	d_i	e_i
J_1	=	(0, 4, 1)	
J_2	=	(0, 4, 1)	
J_3	=	(0, 5, 5)	

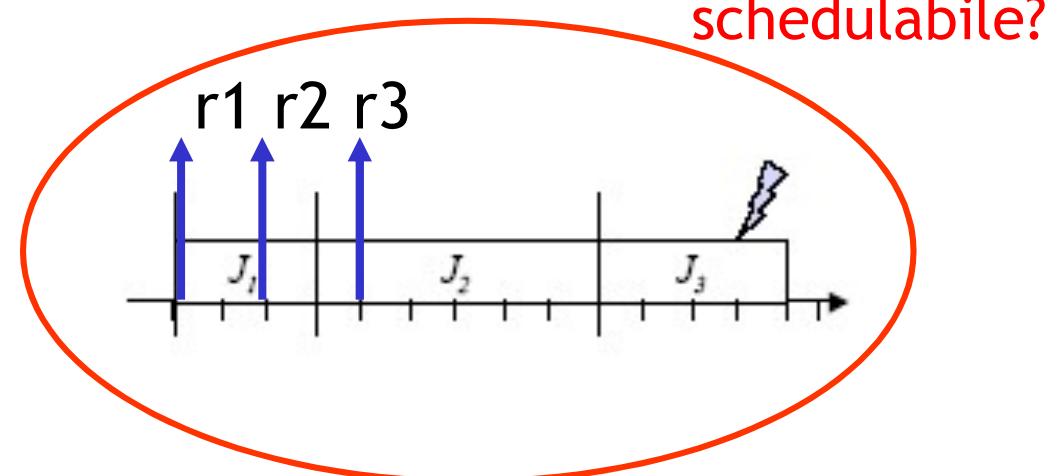




EDF è ottimo solo nelle ipotesi date...

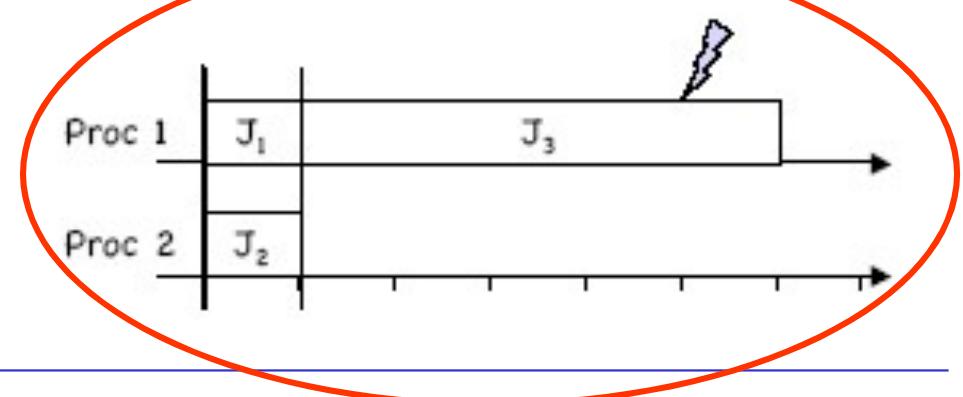
- Caso 1: Preemption non consentita:

	r_i	d_i	e_i
J_1	=	(0, 10, 3)	
J_2	=	(2, 14, 6)	
J_3	=	(4, 12, 4)	



- Caso 2: Sistema multiprocessore:

	r_i	d_i	e_i
J_1	=	(0, 4, 1)	
J_2	=	(0, 4, 1)	
J_3	=	(0, 5, 5)	





Non ottimalità di EDF

- Il caso 1 è schedulabile solo da un algoritmo *non work-conserving*
- Nessun algoritmo priority-driven è in grado di schedulare i job del caso 1
- → In assenza di preemption e con parametri temporali arbitrari, *nessun algoritmo priority-driven* (cioè work-conserving) è *ottimo*



Ottimalità e non ottimalità di EDF

- Q1: in un sistema uniprocesso, in assenza di preemption e con $r_i=0 \forall i$, EDF è ottimo?
- Q2: in assenza di preemption, con istanti di rilascio arbitrari ma multipli di un intervallo elementare, tempi di esecuzione unitari, EDF è ottimo? (Ad es. sistemi sincronizzati con un clock periodico)



Ottimalità e non ottimalità di EDF

- R1: Non c'è mai motivo di avere preemption: non arrivano nuovi job, e tutte le decisioni di scheduling sono prese quando il processore diventa libero -> EDF è ottimo
- L'algoritmo EDF con istanti di rilascio tutti uguali e senza preemption è stato introdotto nei problemi di *job shop scheduling* come *Earliest Due Date* (EDD) (Jackson, 1955)
- R2: Anche in questo caso la preemption non è necessaria



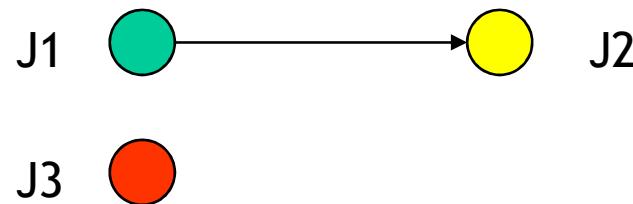
Altri criteri di ordinamento delle priorità

- *Latest Release Time (LRT)*: è l'algoritmo duale rispetto ad EDF.
- Priorità: cresce all'aumentare dell'istante di rilascio
- Schedula i job a partire dall'ultima deadline ed inserisce via via i job con istante di rilascio maggiore
- Tende a lasciare il processore “idle” all'inizio dell'orizzonte temporale: utile per ridurre il tempo medio di risposta dei job soft o non RT
- E' un algoritmo *non work-conserving*



Latest Release Time

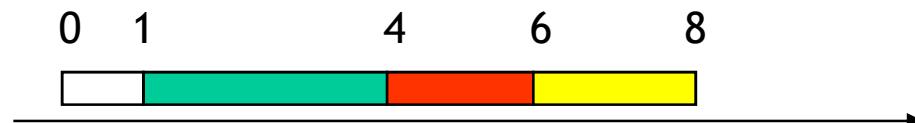
□ Esempio:



	e_i	r_i	d_i
J1	3	0	6
J2	2	5	8
J3	2	2	7

- L'ultima deadline è $t=8$. J2 può essere in esecuzione tra 7 ed 8
- A $t=7$ J2 e J3 possono essere in esecuzione: J2 ha priorità perché $r_2 > r_3$. J2 è quindi in esecuzione tra 6 e 7
- A $t=6$ J1 e J3 possono essere in esecuzione: J3 ha priorità perché $r_3 > r_1$ ed è schedulato tra 6 e 4
- A $t=4$ J1 è schedulato tra 4 e 1

Schedule:





Ottimalità LRT

- Teorema (Ottimalità LRT):

In un sistema *monoprocessore con preemption*, LRT può generare una schedule fattibile per un insieme di job J con istanti di rilascio e deadline arbitrari se e solo se tale schedule esiste.

- Segue dalla dimostrazione di ottimalità per EDF



Limiti della schedulazione LRT

- Attenzione: LRT può essere utilizzato solo se si dispone di *tutte le informazioni* sui job da schedulare per l'orizzonte temporale di interesse
- LRT non può operare dinamicamente perché è *non work conserving*
- Cosa accade, nell'esempio precedente, se all'istante 0 LRT conosce solo J1, l'unico rilasciato? Può attendere a metterlo in esecuzione fino all'istante 3 ($d_1 - e_1 = 3$)? Le fasi di idle possono pregiudicare la schedulabilità successiva!
- work escaping? deferring? pericoloso!



Least Slack Time First (LST) / Minimum Laxity First (MLF)



- In ciascun istante t vale: $\text{slack}_i = d_i - t - e_i(t)$
in cui $e_i(t)$ è il tempo di esecuzione residuo di J_i
- L'algoritmo LST (o MLF) schedula in ciascun istante il job J_i con il minimo slack time, slack_i

- Teorema: Anche l'algoritmo LST/MLF è *ottimo* nelle medesime ipotesi di EDF e LRT



Priority-driven vs. clock-driven

- + Gli algoritmi priority-driven sono più *flessibili* di quelli clock-driven
 - + La loro *realizzazione* (nel caso EDF) si basa su una semplice coda gestita in base alle priorità
 - - Gli algoritmi priority-driven possono esibire un comportamento *non deterministico* se i parametri temporali variano in modo imprevisto
 - - E' più difficile *validare* un insieme di job schedulati in modo priority-driven
 - → alcune tipologie di sistemi hard real-time safety critical sono progettate secondo l'approccio clock-driven
-



Scheduling real-time di task aperiodici

- I vincoli sono tipicamente specificati in termini di makespan dell'insieme o di deadline dei singoli task
- L'algoritmo EDF è ottimo nell'ipotesi di task revocabili e di sistema monoprocesso, con istanti di rilascio e deadline arbitrari
- I vincoli di precedenza possono essere rimossi tramite il calcolo degli istanti di rilascio e delle deadline effettivi
- Per i casi di assenza di preemption o multiprocesso il problema di trovare una schedule fattibile è NP-hard. Gli algoritmi hanno complessità elevata e sono utilizzabili solo off-line



Algoritmo EDD (Jackson, 1955)

- Insieme di n task aperiodici $\{\tau_1, \dots, \tau_n\}$ da eseguire su un processore
- I task sono costituiti da un solo job e sono rilasciati in modo sincrono: $T = \{J_i(C_i, D_i), i=1, \dots, n\}$
- Algoritmo *Earliest Due Date (EDD)*:
“Quando il processore è libero seleziona il task con la deadline relativa minima”
- Caratteristiche:
 - Priorità statica (parametri D_i noti per il rilascio simultaneo dei task)
 - D_i coincide con d_i assumendo $t=0$
 - Non richiede preemption



Algoritmo EDD

- Teorema: Dato un insieme di n task indipendenti, ogni algoritmo che esegue i task *in ordine di deadline non decrescenti* è *ottimo* rispetto al criterio della minimizzazione della massima lateness, L_{\max}
- Se $L_{\max} \leq 0$ tutti i task rispettano la propria deadline
- Test di garanzia: $f_i \leq d_i \quad \forall i$
Ordinando i task in base alle deadline D_i : $f_i = \sum_{k=1,i} C_k$
Il test diviene: $\sum_{k=1,i} C_k \leq D_i \quad \forall i$
- Complessità: $O(n \log n)$ per l'ordinamento dei task, $O(n)$ per garantire il task set



Algoritmo EDF (Horn, 1974)

- EDF estende l'algoritmo EDD considerando *istanti di arrivo arbitrari*
- Algoritmo *Earliest Deadline First (EDF)*:
“In ogni istante esegui il job con la deadline *assoluta* più prossima”
- Caratteristiche:
 - Priorità dinamiche (dipendono dagli arrivi)
 - Preemption consentita ovunque
 - Algoritmo ottimo: minimizza la massima lateness L_{max}
- Complessità: $O(n)$ per inserire un nuovo task pronto in coda, $O(n)$ per garantire un nuovo task -- online



Algoritmo EDF

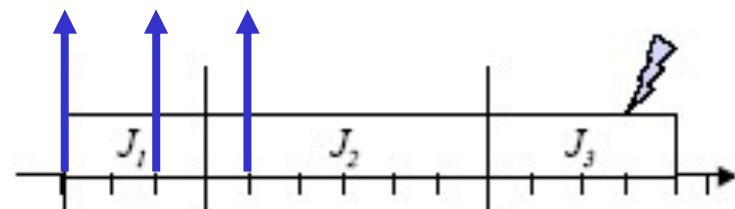
- Test di garanzia: $f_i \leq d_i \quad \forall i$
- Ipotesi: job ordinati in base alle deadline d_i ,
pertanto: $f_i = t + \sum_{k=1,i} C_k(t)$ ove $C_k(t)$ è il
WCET residuo
da cui: $\sum_{k=1,i} C_k(t) \leq d_i - t \quad \forall i$
- Verifica della garanzia *online*: al rilascio di un nuovo job J_i occorre verificare se esso è garantito e se tutti i job J_j accettati in precedenza (con $j > i$, cioè $d_j > d_i$) restano garantiti



Assenza di preemption

- In presenza di istanti di rilascio arbitrari ma senza preemption EDF non è più ottimo:

	r_i	d_i	e_i
J_1	=	(0, 10, 3)	
J_2	=	(2, 14, 6)	
J_3	=	(4, 12, 4)	



- Tuttavia anche in assenza di preemption EDF resta ottimo tra gli algoritmi work-conserving



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Scheduling clock-driven

prof. Stefano Caselli

stefano.caselli@unipr.it



Modello dei task

- L'approccio clock-driven è applicabile quando il sistema è *prevalentemente basato su task periodici* e deterministico
- Al più è possibile gestire *alcuni* task aperiodici o sporadici, che vengono integrati nel *framework deterministico*
- Modello dei task:
 - n task periodici, con n fisso
 - i *parametri* di tutti i task periodici sono *noti a priori*; variazioni nei tempi di inter-rilascio dei task periodici sono trascurabili
 - Ogni job $J_{i,k}$ di τ_i è pronto per l'esecuzione all'istante di rilascio $r_{i,k}$
 - → Non ci sono ulteriori vincoli all'esecuzione dopo il rilascio del job



Task periodico

- Un *task periodico* τ_i è definito da una tupla del tipo:
$$\tau_i = (C_i, D_i, T_i)$$
- Ovvero da:
$$\tau_i = (C_i, D_i, T_i, \Phi_i)$$
- Spesso:
$$\Phi_i = 0 \quad \text{e} \quad D_i = T_i$$
- Nei casi più semplici e frequenti, un task è caratterizzato solo da tempo di esecuzione e periodo: $\tau_i = (C_i, T_i)$ o $\tau_i = (C_i, p_i)$
- *Fattore di utilizzazione:* $U_i = C_i/T_i$



Scheduler statico timer-driven

- La predisposizione off-line di *schedule statiche* permette l'adozione durante la progettazione di *algoritmi sofisticati*, anche complessi
- I periodi in cui il processore è inattivo possono essere resi disponibili per eventuali job aperiodici
- ⇒ Nelle ipotesi date, è *possibile* adottare anche una schedulazione *non work-conserving*
- Quali i potenziali vantaggi/motivazioni?



Scheduler statico timer-driven

- Prima fase: predisposizione off-line di una *schedule statica*, anche con algoritmi sofisticati e di elevata complessità
- E' possibile tenere conto di criteri di priorità specifici, adattando la schedule di conseguenza

- La realizzazione può essere basata su una **tabella** (*scheduling table-driven*) costituita dalle coppie $(t_k, J(t_k))$, ove t_k è l'istante di decisione e $J(t_k)$ è il job da mettere in esecuzione all'istante t_k (oppure ϕ , idle)
- Essendo i task periodici, la tabella avrà un *numero finito di entry*



Scheduler clock-driven

- Input per lo schedulatore: Schedule $(t_k, J(t_k))$, $k=0,1,\dots,N-1$ ($N=\text{num. attivazioni job e intervalli idle}$), iperperiodo H

Task Scheduler:

```
i:=0; k:=0;  
<imposta il timer all'istante  $t_0$ >  
do forever:  
    <attendi il timer interrupt>  
    i:=i+1; // prepara indici e timer per prossimo evento  
    k:=i mod N;  
    <imposta il timer all'istante  $\lfloor i/N \rfloor H + t_k$ >  
    if  $J(t_{k-1})$  è vuoto // esecuzione evento attuale  $k-1$   
    then wakeup(aperiodic)  
    else wakeup( $J(t_{k-1})$ ) // job in tabella, entry  $k-1$   
end  
end Scheduler;
```



Scheduler clock-driven

- Notazione
 - $\lfloor i/N \rfloor$ = div risultato intero della divisione (es: 21 div 10 = 2)
 - $i \text{ div } N$ = numero di iperperiodi già completati
 - t_k tempo relativo all'interno dell'iperperiodo corrente, ottenuto accedendo alla entry k della tabella prememorizzata
- Caratteristiche:
 - Questa struttura di executive consente di gestire schedule non suddivise in frame periodici
 - Ogni t_i è l'inizio di un job o di una fase programmata di idle (in cui saranno eseguiti task aperiodici o in background)



Scheduler clock-driven

- Quali problemi presenta questo approccio?

```
Task Scheduler: //input: Schedule( $t_k, J(t_k)$ ),  $k=0,..N-1$ ;  $H$ 
i:=0; k:=0;
<imposta il timer all'istante  $t_0$ >
do forever:
    <attendi il timer interrupt>
    i:=i+1; // prepara indici e timer per prossimo evento
    k:=i mod N;
    <imposta il timer all'istante  $\lfloor i/N \rfloor H + t_k$ >
    if  $J(t_{k-1})$  è vuoto // esecuzione evento attuale  $k-1$ 
        then wakeup(aperiodic)
        else wakeup( $J(t_{k-1})$ ) // job in tabella, entry  $k-1$ 
    end
end Scheduler;
```



Scheduler clock-driven

- Problemi:
 - Intervalli molto brevi? (es. di idle)
 - Necessità di reimpostare il timer ogni volta, possibile accumulo errori
 - Un solo job per volta, può risultare tabella «lunga»!
 - Controlli di correttezza non integrati: Nuovo job rilasciato? Completato il precedente?
 - Manca politica di gestione errori (overrun e mancati rilasci): Abort? Extend time? Count error? Notify?
- Nel seguito: schema generale di executive per scheduling clock-driven basato su *frame periodici*



Executive ciclico

- I *task periodici* sono una parte importante, spesso prevalente, di molte applicazioni real-time
- Un *executive ciclico* è un programma di controllo che realizza in modo esplicito la *sequenza di esecuzione* (interleaving) di task *periodici* su una singola CPU
- L'interleaving viene realizzato in modo deterministico, in modo che i tempi di esecuzione siano garantiti
- L'interleaving dei job è definito in base ad una *schedule ciclica*



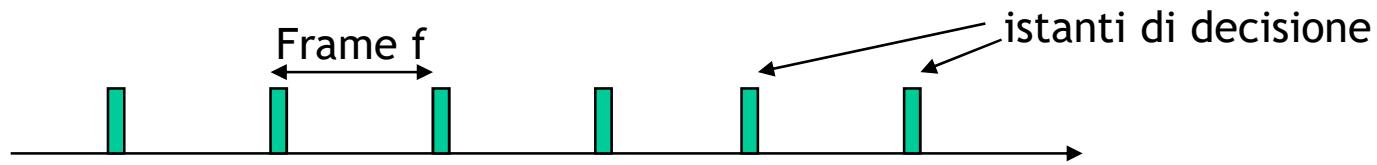
Executive ciclico

-
- Proprietà importanti dell'executive ciclico:
 - Assenza di preemption
 - Non richiede di eseguire preemption a run-time
 - Le commutazioni sono realizzate in istanti pre-pianificati, e quindi “sicuri”
 - Complessità offline
 - È possibile utilizzare algoritmi complessi, che comunque vengono eseguiti offline, per generare una schedule che soddisfi gli obiettivi di performance e di correttezza
 - Rigidità
 - Small change? Do it again
 - Modifiche sostanziali possibili solo offline

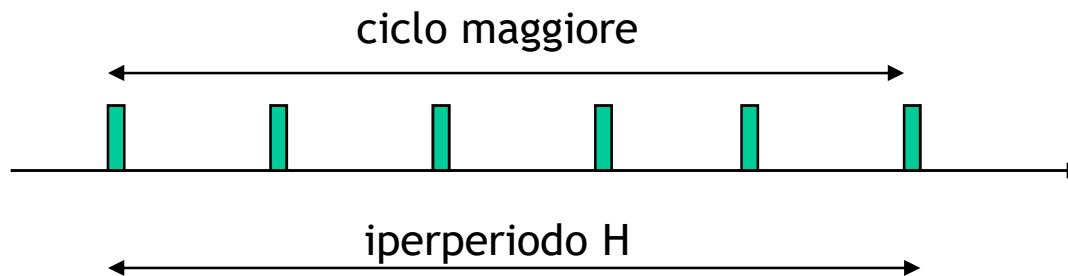


Struttura generale

- Istanti di decisione periodici:



- Le decisioni di scheduling sono prese periodicamente:
 - scelta* del job o dei job da eseguire
 - operazioni di *monitoring* e di *garanzia*
- Ciclo maggiore*: sequenza di *frame* nell'iperperiodo





Il ciclo maggiore

- Una *schedule ciclica* comprende una o più *schedule principali*, che descrivono la sequenza di azioni da eseguire durante un intervallo di tempo prefissato (*ciclo maggiore*)
- Le azioni di una *schedule principale* sono eseguite periodicamente
- La durata del *ciclo maggiore* è pari al minimo comune multiplo dei periodi dei task che fanno parte di ciascuna delle *schedule principali* (*iperperiodo*, H)
- *Schedule principali* diverse corrispondono a *modi* di funzionamento diversi del sistema, tra cui si commuta in corrispondenza a specifici eventi real-time



Frame

- Ciascuna schedule principale è divisa in una o più *schedule secondarie* o *frame*
 - I limiti temporali di un frame corrispondono ad istanti in cui si verifica un interrupt hardware generato da un timer e in cui vengono imposti e verificati i *vincoli temporali*
- A ciascun frame è allocato un intervallo di tempo fisso durante il quale deve eseguire una *sequenza di job* (*scheduling block*)
- Se i job di un frame sono completati in anticipo, il processore attende inattivo o esegue job in background fino all'inizio del successivo frame
- Se i job di un frame non sono completati in tempo, il sistema rileva un errore di *frame overrun*



Funzioni del frame

- gestione del temporizzatore (eventuale)
- dispatching dei job
- abilitazione all'esecuzione di job in background
- verifica del rispetto delle deadline e dei rilasci dei job
- gestione di errori ed eccezioni

- non c'è preemption all'interno di un frame!



Il ciclo minore

- In un executive ciclico i *frame* sono *tutti di uguale durata (ciclo minore)*: la verifica dei vincoli temporali del frame è realizzata mediante la gestione degli eventi generati da un *timer periodico preimpostato*
- Requisito: per la verifica dei vincoli temporali, la *durata massima di ciascun frame* non può essere superiore al periodo minimo (più in generale, alla deadline relativa minima) dei job da eseguire entro il frame



Il ciclo minore

- In un executive ciclico i *frame* sono *tutti di uguale durata (ciclo minore)*: la verifica dei vincoli temporali del frame è realizzata mediante la gestione degli eventi generati da un *timer periodico preimpostato*
- Requisito: per la verifica dei vincoli temporali, la *durata massima di ciascun frame* non può essere superiore al periodo minimo (più in generale, alla deadline relativa minima) dei job da eseguire entro il frame
- A causa del requisito sulla dimensione massima del frame, eventuali job con tempo di esecuzione superiore devono essere suddivisi in *sotto-job*, ciascuno dei quali di durata tale da completare entro un frame --> *job partitioning*



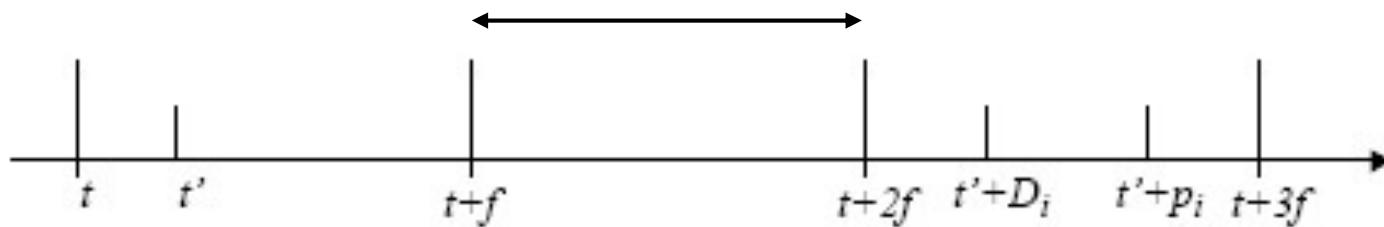
Durata del ciclo minore: requisiti

- $m = \text{durata del ciclo minore}$ o $\text{dimensione del frame}$,
 $M = \text{durata del ciclo maggiore, o iperperiodo}$
- Devono essere soddisfatti simultaneamente i seguenti requisiti:
 - (1) $m \leq D_i \quad \forall i$
 - (2) $m \geq c_i \quad \forall i$
 - (3) $M/m = \lfloor M/m \rfloor$
 - (4) $m + (m - \text{MCD}(m, p_i)) \leq D_i \quad \forall i$
- Nota: Il vincolo (4) susssume (1)



Vincoli sulla dimensione del frame

- La dimensione del frame deve consentire di iniziare e completare ogni job all'interno di un medesimo frame
- m è un divisore intero di H (la condizione m è un divisore intero di almeno un periodo p_i non copre tutti i casi)
- Per monitorare il *rispetto dei vincoli temporali*, i frame devono essere sufficientemente brevi, in modo tale che tra l'istante di rilascio e la deadline di ciascun job si verifichi almeno un frame completo:





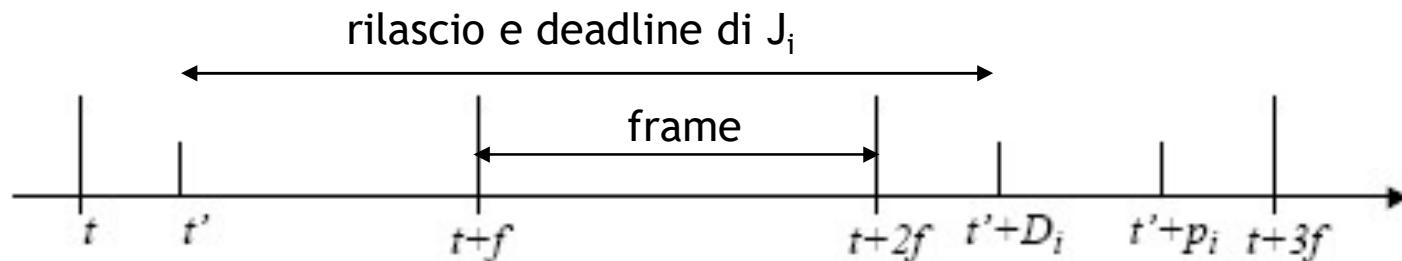
Dimensione del frame

- Queste scelte:
 - evitano la preemption
 - minimizzano la lunghezza totale della schedule
 - assicurano che ogni iperperiodo ospiti un numero intero di frame
 - consentono la verifica del rispetto delle deadline e dei rilasci dei job (ovvero dei vincoli temporali) *all'inizio di ciascun frame*
 - non impongono relazioni di fase specifiche tra i task, che sono spesso impossibili da realizzare



Vincoli sulla dimensione del frame

- Vincolo: $m + (m - MCD(m, p_i)) \leq D_i \quad \forall i$
- L'esecuzione c_i di ciascun job J_i deve essere garantita da un frame (\rightarrow di durata almeno pari al $\max c_i$):
 - il job deve essere attivato dopo l'inizio di un frame
 - il termine del medesimo frame ne deve verificare il completamento prima della deadline
 - ovvero: $m + (m - \min(t' - t)) \leq D_i$





Caso peggiore per l'istante di rilascio

- Il caso peggiore è quello a scostamento minimo tra l'istante di rilascio t' e l'inizio t del frame precedente
- Una proprietà della *teoria dei numeri*:
Date due sequenze periodiche, con periodi X e Y , la distanza minima tra due valori diversi delle sequenze è data dal *massimo comune divisore* dei periodi, $\text{MCD}(X, Y)$
 - Identifica il caso peggiore dal punto di vista del contenimento di una intera esecuzione entro il frame
 - Esclude il caso in cui i valori sono uguali, che sarebbe più favorevole
- Riportiamo il problema ad un problema sugli interi



Esempio

- Calcolo della dimensione del frame per il seguente set di task
 $T_i = (p_i, c_i, D_i)$:
 $T1=(15,1,14) \quad (D_i \neq p_i)$
 $T2=(20,2,26)$
 $T3=(22,3,22)$
- Iperperiodo: $H=660$
- Vincoli:
 - (1) $m \leq D_i \quad \forall i \rightarrow m \leq 14$
 - (2) $m \geq c_i \quad \forall i \rightarrow m \geq 3$
 - (3) m divide $H \rightarrow m=2,3,4,5,6,10,11,12,\dots$
 - (4) $2m - \text{MCD}(m,p_i) \leq D_i \quad \forall i \rightarrow m=2,3,4,5,6$
- → Possibili valori per m : 3,4,5,6



Suddivisione dei job

- Talvolta i parametri non consentono di rispettare tutti i vincoli simultaneamente
- E' possibile rilassare il vincolo (2) suddividendo i job in sotto-job (*job partitioning*)
- Esempio $[T_i = (p_i, c_i, D_i)]$:

$T_1 = (4, 1, 4)$	$(2) \Rightarrow m \geq 5$
$T_2 = (5, 2, 5)$	$(1) \Rightarrow m \leq 4 \quad ???$
$T_3 = (20, 5, 20)$	
- Dove operare il partizionamento? In generale, sarà dipendente dalla applicazione!



Suddivisione dei job

- Se è possibile suddividere T3 in tre job elementari con tempi di esecuzione 1+3+1:

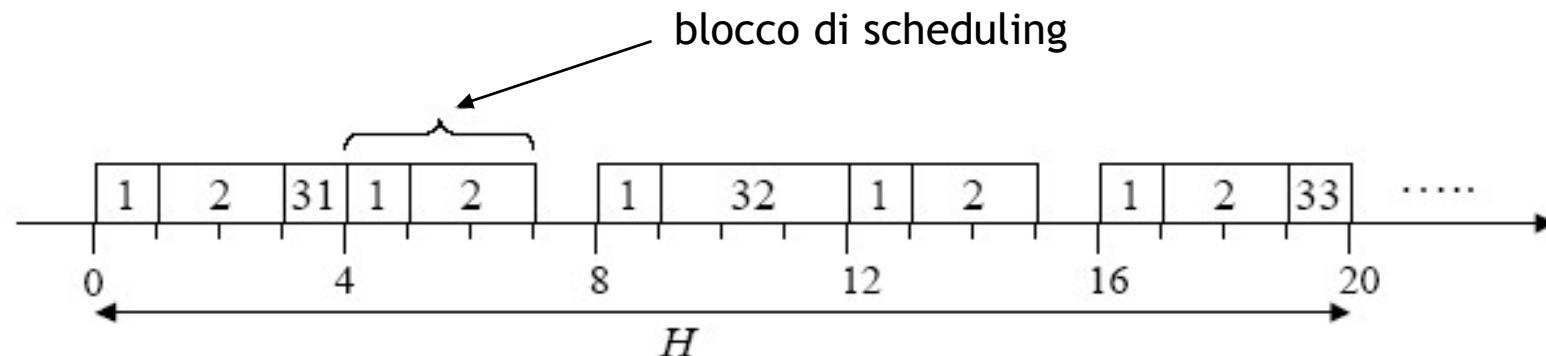
$$T1 = (4, 1, 4)$$

$$T2 = (5, 2, 5) \quad (1) \Rightarrow m \leq 4$$

$$T31 = (20, 1, 20) \quad (2) \Rightarrow m \geq 3$$

$$T32 = (20, 3, 20) \quad \text{Ad es., } m=4$$

$$T33 = (20, 1, 20)$$





Suddivisione dei job

- Se è possibile suddividere T3 in tre job elementari con tempi di esecuzione 1+3+1:

$$T1 = (4, 1, 4)$$

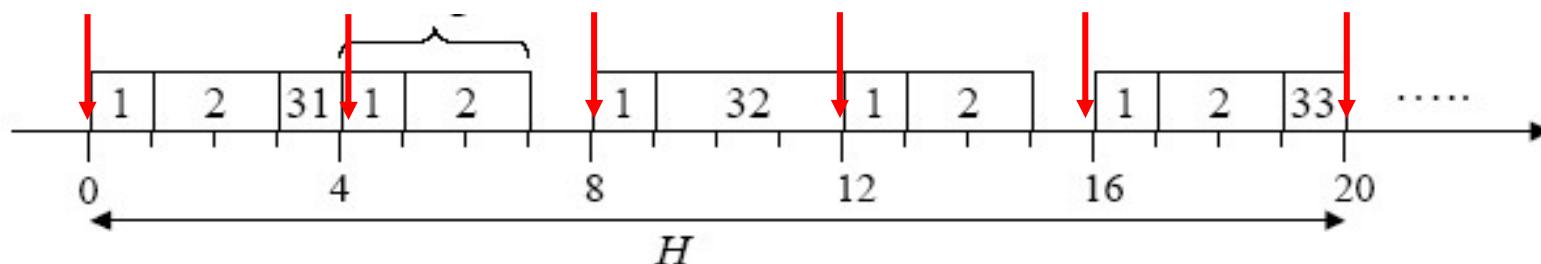
$$T2 = (5, 2, 5) \quad (1) \Rightarrow m \leq 4$$

$$T31 = (20, 1, 20) \quad (2) \Rightarrow m \geq 3$$

$$T32 = (20, 3, 20) \quad \text{Ad es., } m=4$$

$$T33 = (20, 1, 20)$$

Istanti di decisione a inizio frame:
Job frame precedente completati?
Job nuovo frame rilasciati?





Costruzione di una schedule ciclica

- Tre decisioni fondamentali:
 - scelta della *dimensione del frame*
 - partizionamento dei job in sotto-job (slicing)
 - allocazione delle slice nei frame
- In generale, non sono decisioni indipendenti:
 - lo slicing semplifica l'allocazione ma aumenta l'overhead



Partizionamento dei job - L'esempio rivisitato

- Dopo lo slicing di T3:

$$T1 = (4, 1, 4)$$

$$T2 = (5, 2, 5)$$

$$T31 = (20, 1, 20)$$

$$T32 = (20, 3, 20)$$

$$T33 = (20, 1, 20)$$

- Vincoli:

$$(1) m \leq D_i \quad \forall i$$

$$\rightarrow m \leq 4$$

$$(2) m \geq c_i \quad \forall i$$

$$\rightarrow m \geq 3$$

$$(3) m \text{ divide } H$$

$$\rightarrow m=2,4,5,10,20$$

$$(4) 2m - MCD(m, p_i) \leq D_i \quad \forall i$$

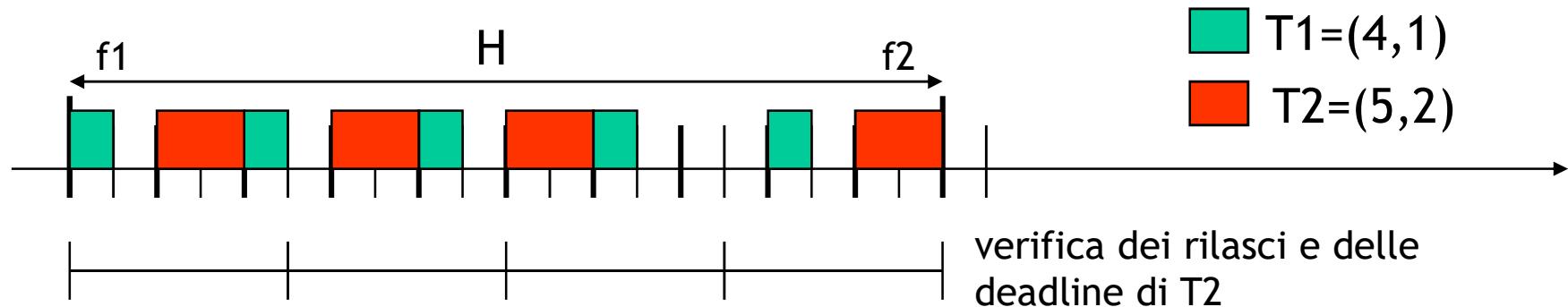
\rightarrow valutiamo solo per $m=4$!

- Con $m=4$: $8\text{-MCD}(4,4) \leq 4$; $8\text{-MCD}(4,5) \leq 5$; $8\text{-MCD}(4,20) \leq 20$
- Il secondo vincolo diventa: $8-1 \leq 5$!! $\rightarrow m=4$ non è una dimensione di frame corretta
- Il vincolo H/m impone di studiare uno slicing per $m=2$



Partizionamento dei job - Esempio (segue)

- Dimensione frame: $m=2$ (ovviamente soddisfa vincoli)
- Occorre indagare un diverso *slicing* di T3
- Allocazione parziale di T1 e T2 sul ciclo maggiore:



- Dopo questa allocazione parziale si vede come solo alcuni partizionamenti di T3 siano possibili
 - Ad es. $(1,1,2,1)$, ma non $(1,2,1,1)$



Partizionamento dei job - Esercizio modificato

- Esercizio: verificare la seguente schedule per l'insieme di task periodici:

$$T_1 = (4, 1, 4)$$

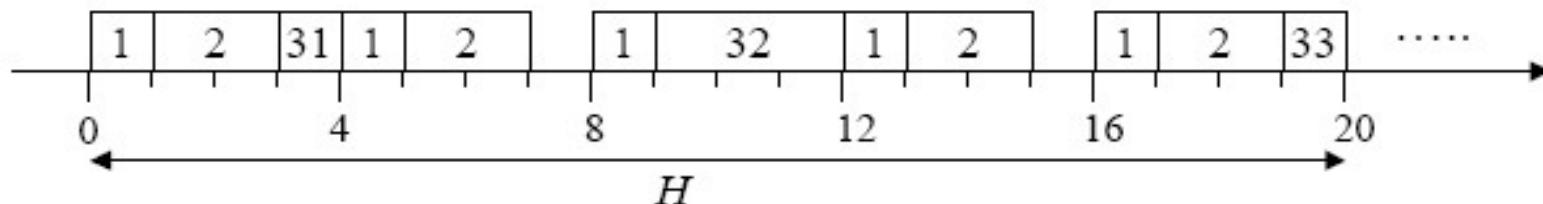
$$T_2 = (5, 2, 7)$$

$$T_{31} = (20, 1, 20)$$

$$T_{32} = (20, 3, 20)$$

$$T_{33} = (20, 1, 20)$$

Vincolo (4): $2m - MCD(m, \pi_i) \leq D_i \forall i$





Osservazioni

- Calcolo della dimensione del frame e partizionamento dei job sono operazioni complesse e non indipendenti
 - Negli esempi abbiamo a che fare con 3 task periodici. Con 20 task?
 - Per minimizzare l'overhead: dimensione di frame massima e minor numero di partizionamenti (*forme di preemption pianificate*)
 - Complessità spostata offline
- Come otteniamo la schedule vera e propria in modo automatico (algoritmo di sintesi)?
- Anche «limitate» variazioni nei parametri dei task possono determinare una modifica radicale della schedule ricavata



Executive ciclico

- Modifica del codice dello scheduler per far sì che le decisioni di scheduling siano prese *solo all'inizio dei frame*
- Ipotizziamo l'esistenza di un timer precaricato che genera un interrupt con periodo f (dimens. frame)
- Input per l'executive:
 - Schedule memorizzata: $L(k)$ per $k=0,1,\dots,F-1$
 - Coda dei job aperiodici
- $L(k)=$ blocco di scheduling



Executive ciclico

```
Task CyclicExecutive:  
t:=0 /* tempo attuale */; k=0 /* frame attuale */;  
CurBlock:=empty;  
do forever:  
    sleep fino al prossimo clock interrupt; /* istante k·f */  
    if <job in CurBlock non completato> invoca FmOverrunHandler;  
    CurBlock:=L(k);  
    k:=k+1 mod F; t:=t+1;  
    if <job in CurBlock non rilasciato> invoca RelErrorHandler;  
    risveglia il server dei task periodici per i job in CurBlock;  
    sleep fino a che il server dei task periodici completa;  
    while <coda dei job aperiodici non vuota>  
        esegui il primo job in coda;  
        rimuovi il job appena completato;  
    end while;  
end do;  
end CyclicExecutive;
```



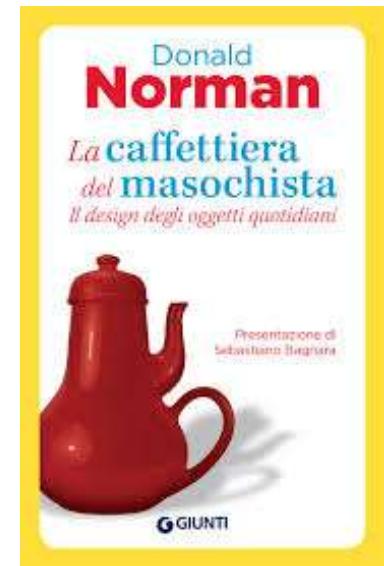
Osservazioni

- Importante: il timer *non* viene ricaricato intervallo per intervallo; --> misura il tempo in modo regolare senza derive
- Verifiche dei vincoli temporali su rilasci e deadline integrate negli istanti di decisione
- Ogni frame può eseguire uno scheduling block
 - tipicamente l'executive assegna a un thread (periodic server) o a thread specifici l'esecuzione dei job/slice dello scheduling block
- Per utilizzare l'idle time in modo controllato si può prevedere un `aperiodic_task_server()`, che esegua a priorità inferiore rispetto all'executive e al `periodic_task_server()`

Integrazione di carico aperiodico e sporadico in scheduling clock-driven

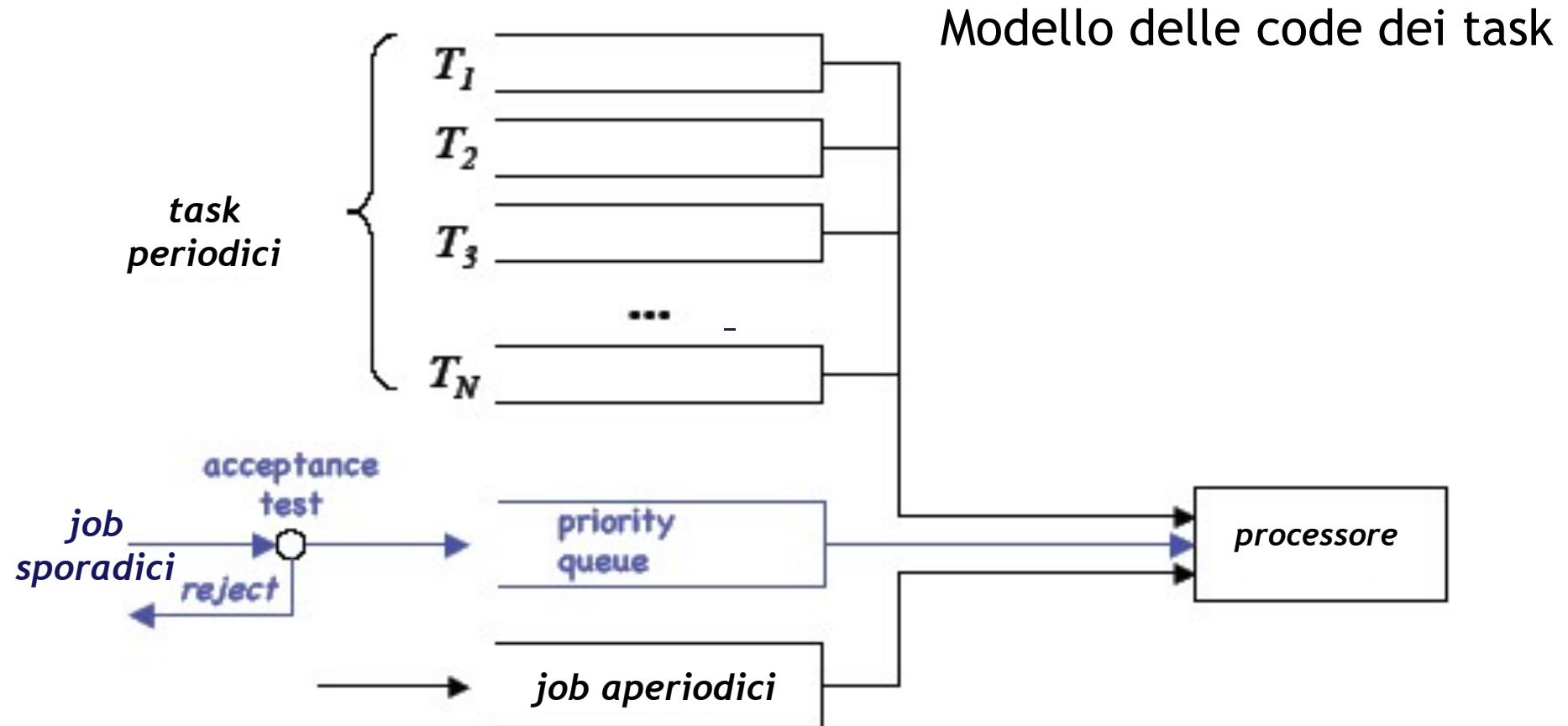


- Ipotesi:
- Il carico associato ai task periodici hard real-time è prevalente
 - altrimenti l'approccio clock-driven non è quello appropriato!
- Estensione:
 - integrazione di task aperiodici -> soft real-time
 - Integrazione di task sporadici -> hard real-time;
richiede *accettazione*





Scheduling clock-driven di task periodici con integrazione di job sporadici e aperiodici





Job aperiodici

- Spesso schedulati in background, non presentano deadline hard
- I job aperiodici sono attività di risposta ad *eventi esterni*
- Obiettivo progettuale: riduzione del *tempo di risposta* medio
- Approccio: *Slack stealing* (Lehoczky, 1987)



Slack stealing

- *Slack stealing*: esecuzione dei job aperiodici prima dei job periodici quando possibile
- Ipotesi: ogni job o slice periodico è schedulato in un frame che termina entro la sua deadline
 - x_k : tempo totale allocato ai job nel frame k
 - $f - x_k$: slack time all'inizio del frame k
- Dopo che y unità di slack time sono state già utilizzate, è possibile eseguire job aperiodici nel frame fino a quando vale: $f - x_k - y > 0$



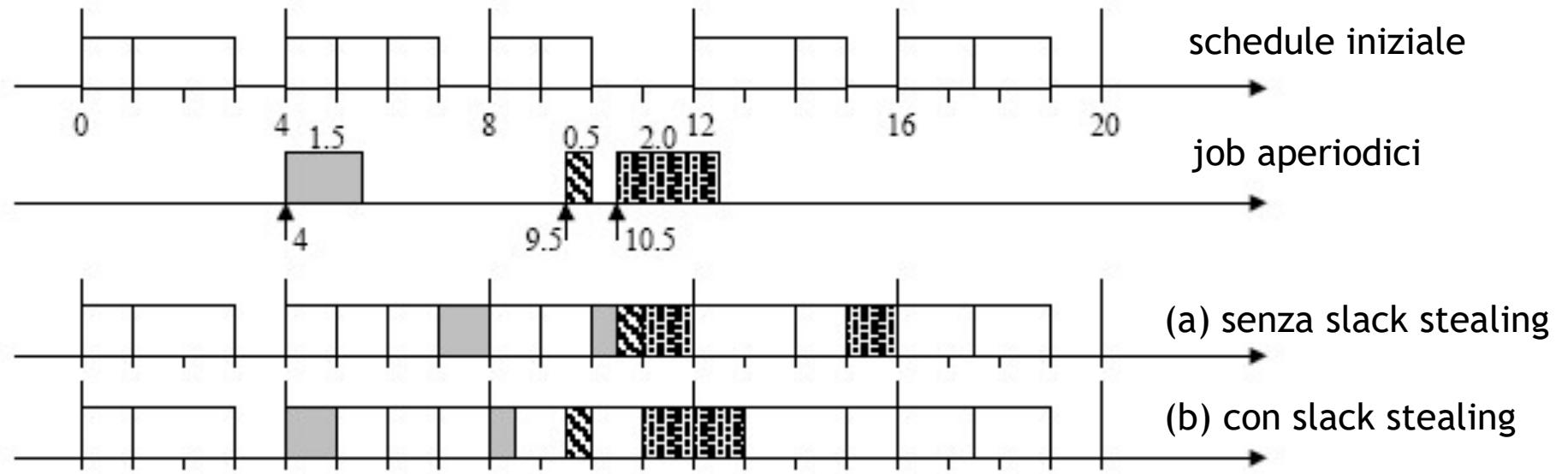
Slack stealing

- Se i job aperiodici si esauriscono con $f - x_k - y > 0$, l'executive pone in esecuzione il *periodic server* per il blocco di scheduling $L(k)$
- Al termine di ogni job in $L(k)$, se $f - x_k - y > 0$ l'executive verifica l'eventuale presenza di nuovi job aperiodici in coda
- Lo slack è rimasto invariato (o è aumentato se $C_i(k) < WCET$)
- Job aperiodici non completati all'interno del frame vengono nuovamente inseriti nella coda dei job aperiodici → preemption



Slack stealing

- Esempio:



Tempi di risposta medi:

$$T_a = (6.5 + 1.5 + 5.5) / 3 = 4.5$$

$$T_b = (4.5 + 0.5 + 2.5) / 3 = 2.5$$



Slack stealing

- Realizzazione:
 - lo slack time iniziale in ciascun frame può essere precalcolato e memorizzato in tabella con $L(k)$
 - l'executive deve tenere traccia del tempo dedicato ai job aperiodici ed aggiornare lo slack time disponibile
 - occorre un timer da settare allo slack time disponibile all'inizio del frame
- Problemi:
 - timer con adeguata risoluzione;
 - se lo slack time è scarso, il timer è sufficientemente accurato? vale la pena mettere in esecuzione il job?
 - in pratica applicabile con slack e parametri ≥ 10 ms



Job sporadici

- Job con caratteristiche asincrone e istanti di rilascio non prevedibili ma che *richiedono garanzie real-time*
- Per poter fornire garanzie, è necessario formulare alcune *ipotesi sulle caratteristiche dei job*:
 - tempo minimo di interarrivo noto,
 - tempo di esecuzione WCET noto o dichiarato all'arrivo insieme alla deadline,
 - eventuale conoscenza a priori del numero e delle caratteristiche di tutti i possibili job sporadici
- In assenza di queste conoscenze a priori, è necessario prevedere un modulo di accettazione: job sporadici potranno essere rifiutati; quelli ammessi nel sistema saranno garantiti



Job sporadici in sistemi clock-driven

- Situazione frequente:
 - uno o pochi job sporadici con caratteristiche note a priori $J_s(D,e)$
 - i job possono essere implicitamente garantiti dalla presenza di sufficiente slack time, noto, nell'iperperiodo H
- Se i job sporadici non hanno caratteristiche note o sono numerosi:
 - (a) approccio sbagliato?
 - (b) è necessario prevedere accettazione
 - il job non garantito viene rifiutato;
 - può essere risottoposto con requisiti inferiori di QoS (es. D maggiore) oppure come job aperiodico eseguibile in modo «best effort»

Job sporadici



- Problema: gestione dei job sporadici nel caso più generale
- Idea base per la gestione dei job sporadici nell'ambito di uno scheduling clock-driven:
 - Usiamo la *trama sincrona* dei frame per garantire le deadline dei job sporadici

Job sporadici

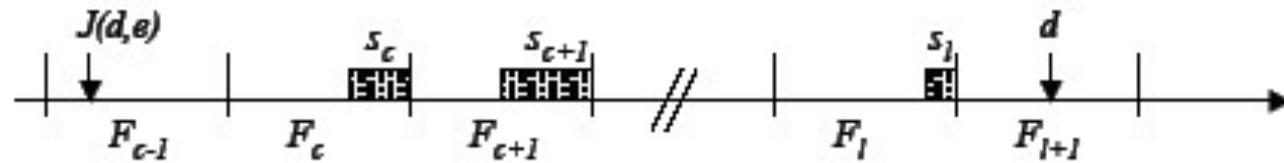


- Caratteristiche: *deadline hard*, istanti di rilascio e tempi di esecuzione non noti a priori, tempo massimo di esecuzione noto all'istante di arrivo, revocabili
- Richiedono *test di accettazione*: il job sporadico viene accettato se è schedulabile e non compromette la schedulabilità dei job periodici o di altri job sporadici già accettati
- $Js(d,e)$ è *schedulabile* se $S_c(t,l) \geq e$, ove $S_c(t,l)$ è lo slack corrente totale dall'istante attuale t fino al frame l che precede la deadline d ($l+1$ termina dopo d)
- La *garanzia* di Js (hard), richiede che ne venga verificato il completamento al più entro la deadline



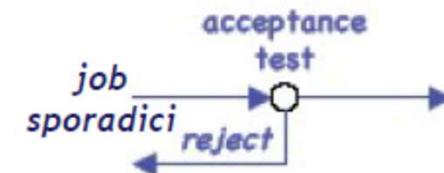
Job sporadici

- test di accettazione:



```
if  $S_c(t, l) < e$  then reject job;  
else  
    accept job;  
    schedula esecuzione;  
end;
```

- Più job sporadici in attesa di accettazione possono essere valutati *in ordine EDF*





Scheduling dei job sporadici accettati

- Scheduling statico:
 - Scheduling nel frame corrente di una slice (più grande possibile) del job accettato
 - Scheduling della parte rimanente il più tardi possibile, eventualmente in più slice
- Meccanismo:
 - Le slice del job accettato sono appese, nei frame in cui vengono schedulate, alla lista dei job periodici
- Problema: arrivo di successivi job sporadici
- Alternative:
 - Rescheduling all'arrivo di nuovi job
 - Scheduling priority-driven dei job sporadici (ad es. EDF)



Scheduling EDF dei job sporadici accettati

- Soluzione realistica: valutazione e accettazione dei job sporadici all'inizio del frame, in modo EDF
- La coda dei job sporadici accettati viene mantenuta ordinata in modo EDF
- I job sporadici possono essere eseguiti, ad esempio, al termine dei job periodici del frame (non c'è vantaggio ad anticipare)
- In presenza di job sporadici e aperiodici devono essere favoriti quelli sporadici, tuttavia è possibile aggiungere lo slack stealing per gli aperiodici
- Lo scheduling dei job sporadici accettati in modo EDF è ottimo *tra gli algoritmi che eseguono il test di accettazione all'inizio del frame*



Realizzazione del test di accettazione

- Ipotesi: test di accettazione *all'inizio del frame (→ EDF ciclico)*
- Test per il job $Js(d,e)$:
 - (1) Determinare se lo slack time nei frame che precedono d è sufficiente: $S_c(t,l) \geq e$
 - (2) Verificare che altri job sporadici già accettati con deadline successiva a d non completino in ritardo
- Se una delle due condizioni è falsa → reject $Js(d,e)$
- E' utile disporre dello slack corrente totale tra ogni coppia di frame i e k , precalcolandolo per i job periodici (richiede una matrice triangolare, $O(|F|^2)$)
- Se i e k appartengono a due cicli maggiori diversi j e j' :
$$S(i+(j-1)F, k+(j'-1)F) = S(i,F) + S(1,k) + (j'-j-1)S(1,F)$$



Realizzazione del test di accettazione

- Allo slack time precalcolato in base ai job periodici occorre sottrarre la quota ancora da eseguire dei job che avranno priorità sul job corrente nell'ordinamento EDF
- Se $Js(d,e)$ è accettato, i job J_k con deadline $d_k > d$ saranno ritardati; deve valere: $s_k = s_k - e \geq 0 \quad \forall k$ tale che $d_k > d$
- Prima di accettare Js occorre quindi verificare che questi job completino ancora entro le proprie deadline
- Al crescere del numero di job sporadici, il sistema diventa complesso e perde i vantaggi di semplicità e affidabilità dell'approccio clock-driven
- → in pratica si applica per pochi job che non rientrano nel paradigma periodico



Problemi realizzativi dell'approccio clock-driven: Frame overrun

- Possibile? I task non erano stati accuratamente garantiti con sofisticate analisi offline?
- Cause: WCET imprecisi, guasti transienti
- Possibili politiche per overrun sporadici:
 - abortire il job all'inizio del nuovo frame
 - preemption (se non in sez. critica); la parte residua è eseguita come job aperiodico nei frame successivi
 - continuare l'esecuzione; i job del frame successivo sono ritardati; ad es. consentire al più lo slack time del frame successivo
 - mix; comunque tracciare evento
 - con overload prolungato, garantire (k su m) job per uno o più task?
- La politica più adatta dipende dalla funzione valore dei task per l'applicazione specifica

Problemi realizzativi dell'approccio clock-driven: Cambiamenti di modo



- Per riconfigurazione del sistema
- Normalmente è necessario un lasso di tempo per caricare nuove tabelle e codice dei job e per allocare memoria
- → *mode change job* ! Può essere hard o soft
 - aperiodic mode change
 - sporadic mode change
- Quando intervenire (sui singoli job o su insiemi di job)?
 - al termine del ciclo maggiore
 - al termine del frame
 - immediatamente (abort del job in esecuzione e dei job partizionati)

Problemi realizzativi dell'approccio clock-driven: Cambiamenti di modo



- Job di task appartenenti ad entrambi i modi restano in esecuzione
- E' necessario conoscere la rilevanza dei job aperiodici e sporadici nel nuovo *modo*
- Ad es. (ipotesi): i job aperiodici possono essere abortiti, mentre gli sporadici vanno completati

- In generale esiste un problema di *transitorio*; obiettivo: minimizzazione della durata del transitorio mantenendo la consistenza del sistema



Pregi e difetti dello scheduling clock-driven

- Vantaggi:
 - + semplicità concettuale
 - + assenza di anomalie
 - + predicitività di funzionamento
 - + minimo overhead
 - + assenza o limitazione di preemption non pianificate

Pregi e difetti dello scheduling clock-driven



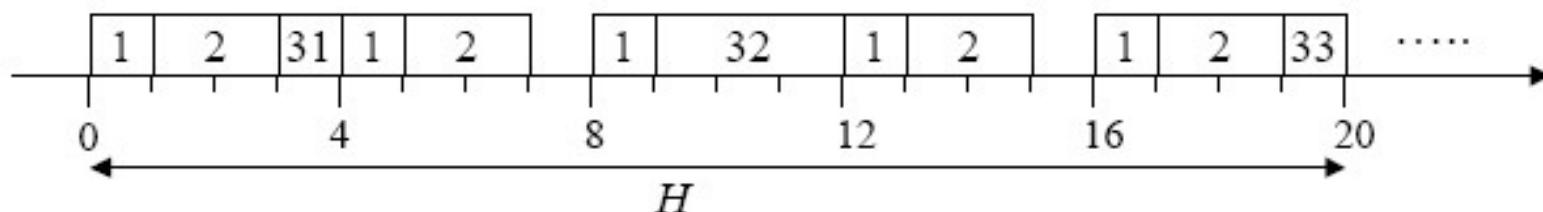
- Limitazioni:
 - rigidità
 - difficoltà di modifica e manutenzione
 - richiede conoscenza a priori (istanti di rilascio dei task periodici, combinazioni di task periodici nei diversi modi, etc.)
 - integrazione problematica di componenti di carico dinamiche



Manca ancora qualcosa: Come allocare i job ai frame?

- L'allocazione manuale dei job ai frame è plausibile solo per pochi task periodici
- In generale si tratta di trovare una schedule che assegna i job ai frame compatibili con i vincoli temporali
- Il problema è *trattabile* e ammette infinite soluzioni se la preemption è ovunque consentita
- Esempio:

$$T_1 = (4, 1, 4) \quad T_2 = (5, 2, 7) \quad T_{31} = (20, 1, 20) \quad T_{32} = (20, 3, 20) \quad T_{33} = (20, 1, 20)$$

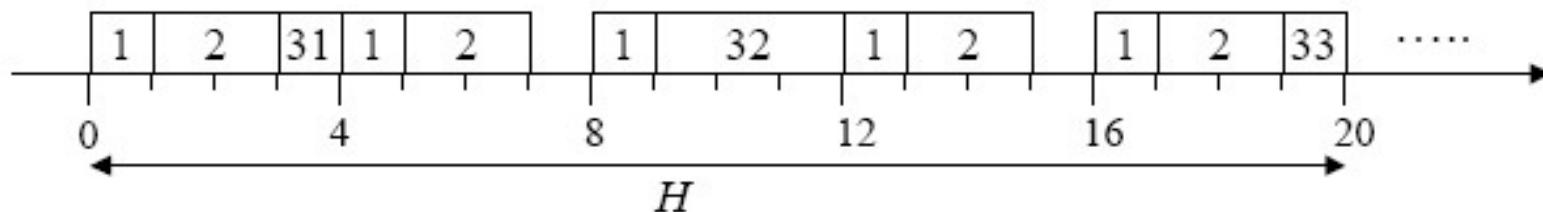




Manca ancora qualcosa: Come allocare i job ai frame?

- L'allocazione manuale dei job ai frame è plausibile solo per pochi task periodici
- In generale si tratta di trovare una schedule che assegna i job ai frame compatibili con i vincoli temporali
- Il problema è *trattabile* e ammette infinite soluzioni se la preemption è ovunque consentita Q: Che succede se invece la preemption non è consentita? Che tipo di problema diventa?
- Esempio:

$$T_1 = (4, 1, 4) \quad T_2 = (5, 2, 7) \quad T_{31} = (20, 1, 20) \quad T_{32} = (20, 3, 20) \quad T_{33} = (20, 1, 20)$$



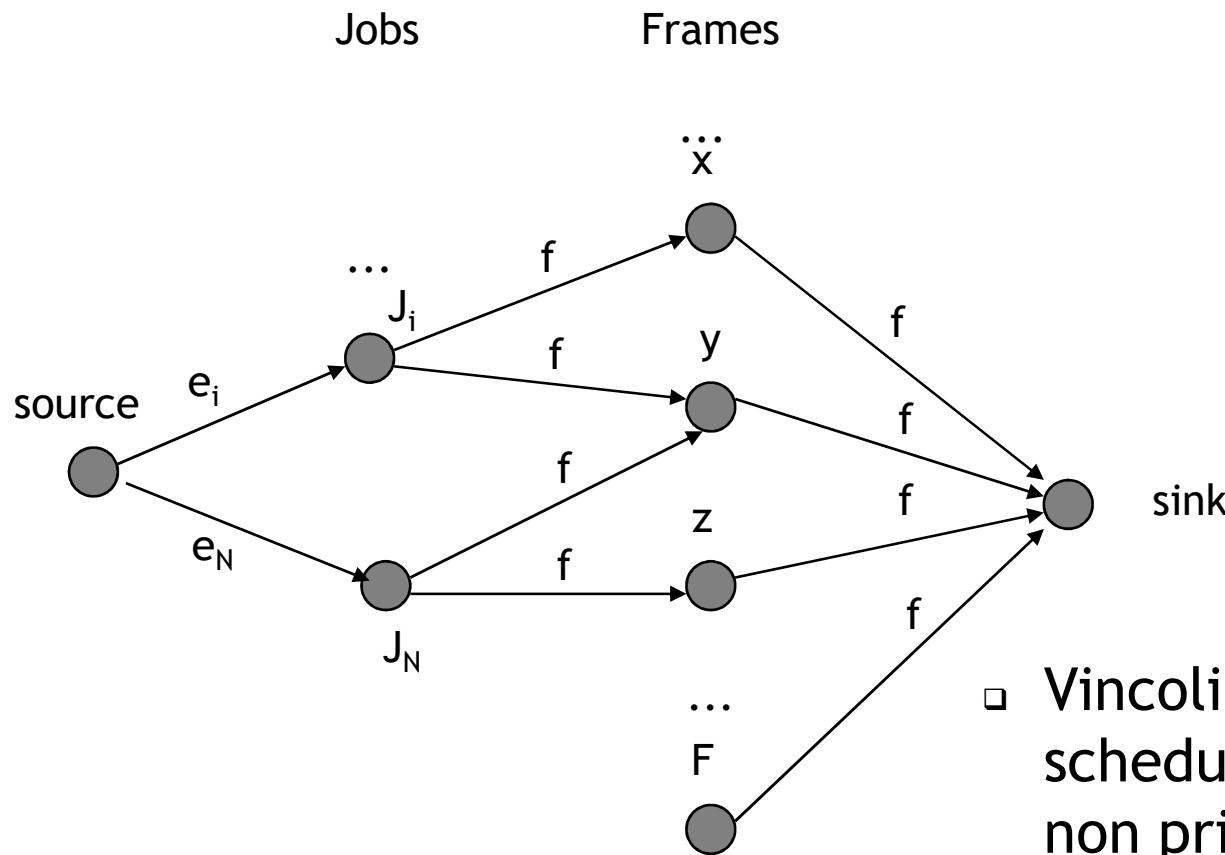


Algoritmo per sintesi di schedule statiche

- Nel caso generale il problema è *NP-hard*
- Se la preemption è consentita ovunque diventa polinomiale e può essere risolto con un algoritmo che calcoli il *flusso massimo su un grafo*
- Struttura del *grafo*:
 - un vertice per ogni job o slice $J_i(d_i, e_i)$ all'interno del ciclo maggiore: $\{J_i, i=1, \dots, N\}$
 - un vertice per ogni frame: $\{f_j, j=1, \dots, F\}$
 - vertici *source* e *sink* per il calcolo del flusso max
 - lati (J_i, f_j) di capacità f se J_i può essere schedulato in f_j
 - lati da source a J_i di capacità e_i
 - lati da f_j a sink di capacità f



Struttura del grafo di flusso



- Vincoli: un job o slice può essere schedulato in un frame che inizia non prima del suo istante di rilascio e termina non oltre la sua deadline

Algoritmo per sintesi di schedule statiche / 1



- *Inizializzazione:*
- Calcola tutte le possibili dimensioni di frame che soddisfano i vincoli:
 - $\lfloor H/f \rfloor - H/f = 0$
 - $2f - MCD(f, p_i) \leq D_i \quad \forall i$
- Il vincolo $f \geq e_i \quad \forall i$ può essere ignorato (per possibile partizionamento dei job)

- *Iterazione:*
- A partire dal valore massimo di f , per la dimensione di frame corrente costruisci il grafo di flusso ed esegui un algoritmo per il calcolo del flusso massimo



Algoritmo per sintesi di schedule statiche /2

- Il flusso massimo possibile, in base alla struttura del grafo, è soggetto al vincolo $\Phi \leq \sum_{i=1,N} e_i$
- La *schedule esiste* (e viene costruita dall'algoritmo) se il flusso massimo calcolato è uguale alla somma dei tempi di esecuzione: $\Phi_{\max} = \sum_{i=1,N} e_i$
- Se un job è eseguito su più frame, significa che esso deve essere partizionato in sottojob
- Se il flusso massimo è inferiore a Φ_{\max} non esiste una schedule fattibile per il valore di f corrente e occorre indagare altre dimensioni di frame

Algoritmo per sintesi di schedule statiche



-
- *Flusso massimo del grafo*: formulazione e risoluzione come problema *LP* o risoluzione tramite *algoritmo dei cammini aumentati*



Algoritmo dei cammini aumentati

-
0. Trasforma il grafo orientato in non orientato; la capacità è annotata in prossimità del vertice di partenza del lato.
 1. Trova un cammino che aumenta il flusso totale, cioè un cammino diretto tra source e sink tale che ogni lato del cammino abbia una capacità > 0 .
 - 1.b Se il cammino non esiste, il flusso identificato è già ottimo.
 2. Identifica il lato del cammino con capacità residua minima c^* . Aumenta il flusso del cammino di c^* .
 3. Diminuisci di c^* la capacità residua di ogni lato lungo il cammino. Aumenta di c^* la capacità residua di ogni lato del cammino in direzione opposta. Ritorna al passo 1.



Algoritmo dei cammini aumentati

□ Al termine dell'algoritmo:

- se $\Phi_{\max} = \sum_{i=1,N} e_i$ l'algoritmo ha determinato una schedule fattibile
- l'insieme dei *flussi dei lati dai vertici che rappresentano i job ai vertici che rappresentano i frame* costituisce una *schedule fattibile preemptive*



Sintesi di schedule statiche

- Esempio: $[T1=(4,1), T2=(5,2,7), T3=(20,5)]$
- Troviamo (ad es.) la schedule: $H=20$, $m=4$, con
 $f1=[J11(1),J21(2)]; f2=[J12(1),J311(3)]; f3=[J13(1),J22(2)];$
 $f4=[J14(1),J23(2),J312(1)]; f5=[J15(1),J24(2),J313(1)]$
- Altri esempi per esercizi:
 $[T1=(4,3), T2=(6,1.5)]$
 $[T1=(4,1), T2=(5,2,5), T3=(20,5)]$
- Esempio Liu, p. 86:
 $[T1=(4,1), T2=(5,1.8), T3=(20,1), T4=(20,2)]$



Foglio di lavoro



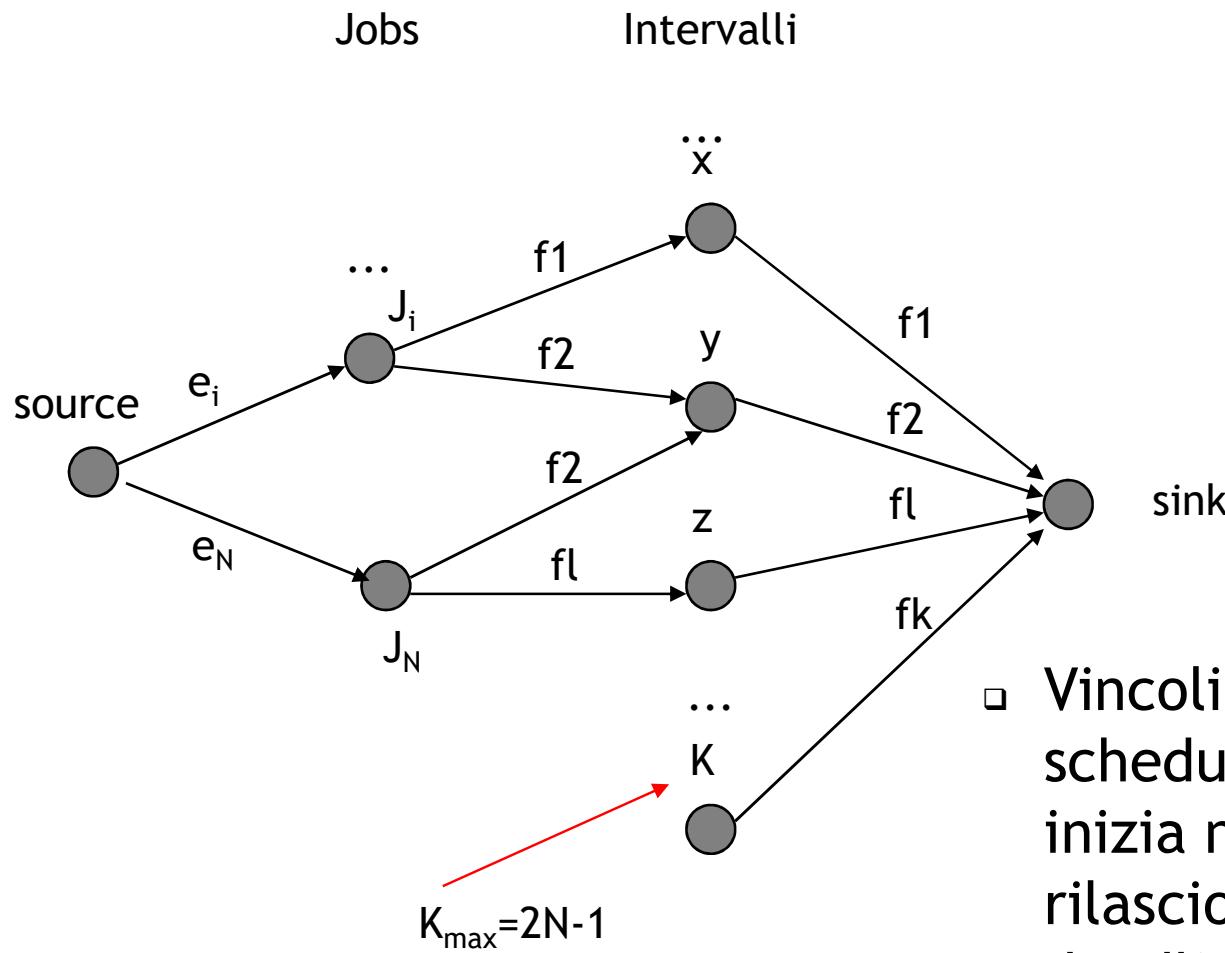
Foglio di lavoro/2



Sintesi di schedule per task non periodici

- La formulazione tramite calcolo del flusso massimo del problema di scheduling può essere generalizzata per ricavare schedule di *job aperiodici indipendenti* con istanti di rilascio e deadline arbitrari
- Gli istanti di rilascio e le deadline di N job dividono il tempo in al più $2N-1$ *intervalli*
- Nel grafo è sufficiente sostituire i vertici dei frame con vertici I_j che rappresentano ciascuno degli intervalli
- La capacità entrante ed uscente di ciascun vertice I_j è pari alla dimensione dell'intervallo
- Se $\Phi = \sum_{i=1, N} e_i$ la schedule *preemptive* rappresentata dai flussi è fattibile

Grafo di flusso per schedulazione aperiodica



- ❑ **Vincoli:** un job o slice può essere schedulato in un *intervallo* che inizia non prima del suo istante di rilascio e termina non oltre la sua deadline



Esercizio

-
- Per il seguente insieme di job aperiodici, determinare una schedule mediante algoritmo dei cammini aumentati
 - $J_i = (d_i, c_i, r_i)$: $J_1 = (12, 3, 0)$, $J_2 = (8, 3, 0)$, $J_3 = (14, 4, 6)$, $J_4 = (6, 2, 2)$



Assegnamento

- Per i job dell'esercizio precedente ricavare anche le schedule prodotte dagli algoritmi EDF e LRT
- Ricercare le schedule prodotte dagli algoritmi FIFO, SJF, LJF (con preemption), verificandone la fattibilità

- Inviare soluzione in unico file pdf entro Lunedì 20/3 ore 12.30, con oggetto email:
SORT2023 - Esercizi Lezione 17/3



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Scheduling di job aperiodici e sporadici

In scheduling priority-driven
di task periodici



Introduzione

- Scenario realistico: compresenza di task “misti”
- Task:
 - Periodici -> (generalmente) hard RT
 - *Aperiodici* (in senso lato):
 - Hard RT (task sporadici)
 - Soft RT
 - Non RT
- I task periodici sono in genere gestiti con uno degli algoritmi già presentati, indipendentemente dalla presenza di task aperiodici
- Tipicamente: RM o DM se a priorità statica, EDF se a priorità dinamica



Introduzione

- Ipotesi:
 - Task periodici rilasciati simultaneamente all'istante $t=0$
 - Task e job aperiodici rilasciati in istanti arbitrari, non noti a priori
- Obiettivo:
- Garantire i task periodici senza penalizzare eccessivamente i task aperiodici:
 - Per i task aperiodici SoftRT e NonRT -> minimizzare il tempo medio di risposta
 - Per i task aperiodici HardRT (task sporadici) -> *garantirne* il completamento entro la deadline previa accettazione



Introduzione

- Ipotesi generali: rilasciamo la sola ipotesi di periodicità di tutti i task, mentre manteniamo le altre →
 - Un solo processore
 - Preemption possibile ovunque, sia per task periodici che per task sporadici e aperiodici
 - Task periodici e job non periodici tutti indipendenti tra loro

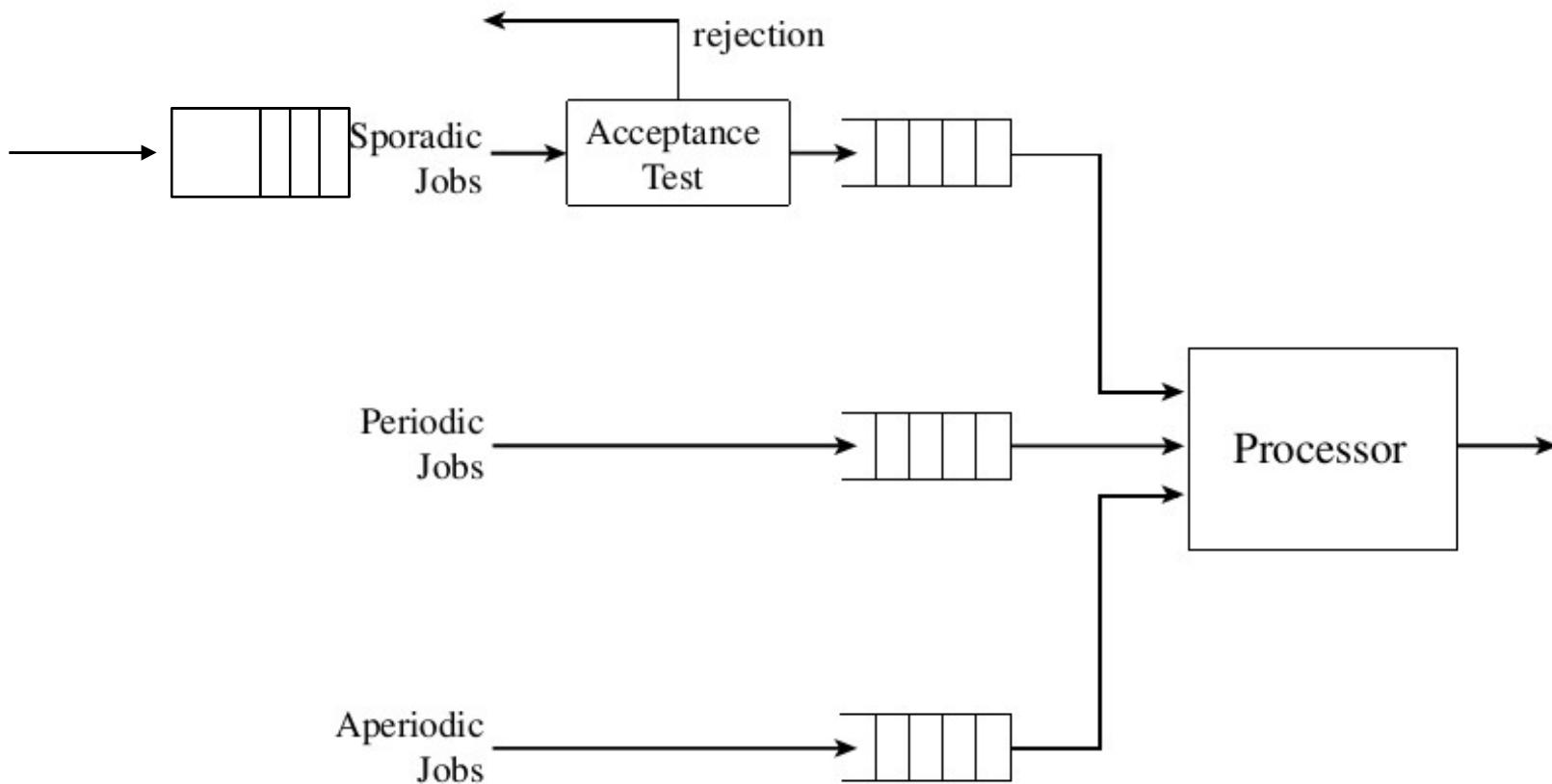


Garanzia per carico non periodico

- Sul singolo job: → on-line
- Sull'intero task *non periodico* con istanze ripetute:
 - Per essere data in sede di prima attivazione occorre conoscere il tempo minimo tra due rilasci → task *sporadici*
 - Si assume talvolta che la deadline relativa sia il tempo minimo tra due rilasci



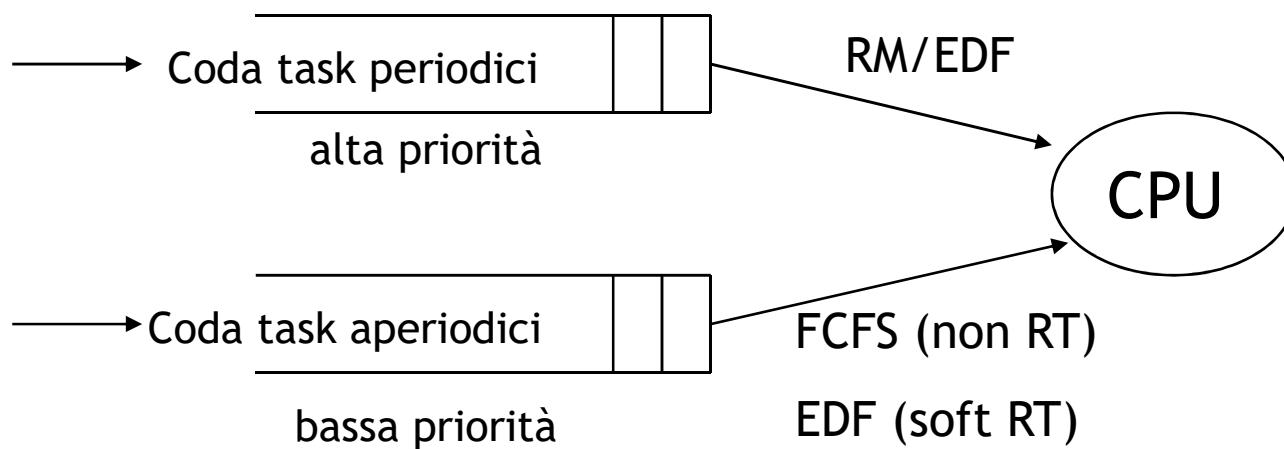
Architettura con code di priorità





Schedulazione in background

- I task aperiodici sono schedulati quando *il processore è libero* e non ci sono job periodici o sporadici pronti
- Problemi: possibile *starvation* in presenza di carico elevato dei task periodici, tempi di risposta elevati

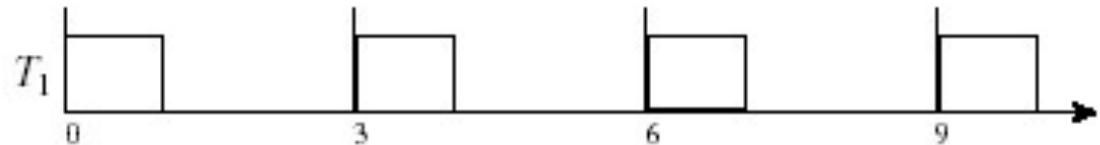


- Applicabile anche con altri algoritmi per i task periodici
- Strategie indipendenti per le due code



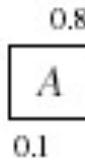
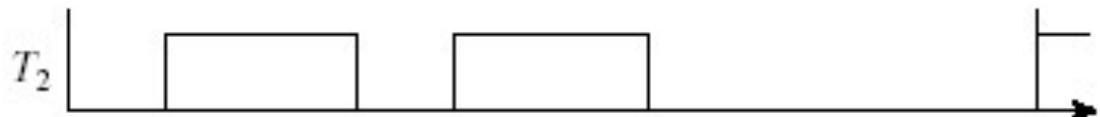
Tempo di risposta con schedulazione in background

$T_1 = (3, 1)$



$T_2 = (10, 4)$

A =aperiodico



- A rilasciato in $t=0.1$, con $C_a=0.8 \rightarrow f_a=7.8$ e $t_{risp}=7.7$
- Prestazioni basse, potrebbe essere $t_{risp}=0.8!$



Schedulazione in background

- Idle time disponibile per l'esecuzione aperiodica in ogni iperperiodo H :

$$\Phi = (1 - U) H$$

- Sia T un insieme di task periodici, con utilizzazione complessiva U e iperperiodo H . Un job aperiodico hard real-time $Ja(Ca, Da)$, ove Da è la deadline relativa del job, è garantito se

$$\lceil Ca / \Phi \rceil H \leq Da$$

- E' una condizione sufficiente
- $\lceil Ca / \Phi \rceil$ è il numero di iperperiodi necessari per soddisfare la richiesta Ca
- Utile solo per $Da \geq H$



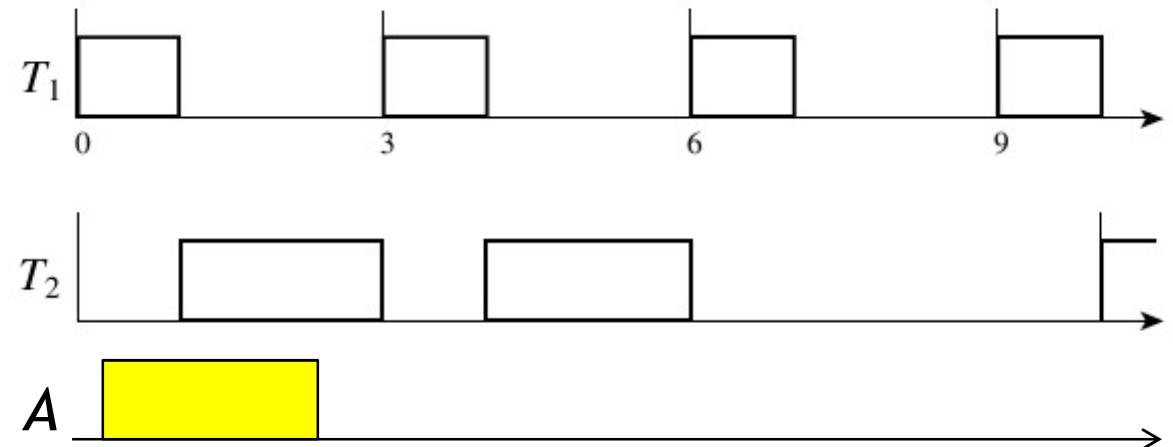
Schedulazione in background

- Per m job aperiodici J_1, \dots, J_m hard real-time, ordinando l'insieme di task aperiodici per deadline assolute crescenti (\rightarrow EDF), l'insieme è garantito al tempo t se:
$$\forall h=1, \dots, m \quad t + \lceil S_h / \Phi \rceil H \leq d_h$$
- ove: $S_h = \sum_{i=1,h} C_i(t)$, $C_i(t)$ = tempo residuo di esecuzione del job J_i all'istante t , d_h = deadline assoluta di J_h
- Risultato utile, come upper bound del tempo di risposta, anche per task soft real-time e non real-time



Schedulazione interrupt-driven

- Il job aperiodico viene posto in esecuzione non appena rilasciato, interrompendo l'esecuzione dei task periodici
- Minimizza il tempo di risposta dei job aperiodici
- Problema: possibili deadline miss per task periodici!
- Es.: $T_1=(3,1)$, $T_2=(10,4)$, $A=(R_A=0.1, E_A=2.1)$
- Deadline miss:
 - J11
 - J21





Schedulazione interrupt-driven

- In generale *non appropriata nei sistemi real-time*, se non per garantire uno o pochi task aperiodici/sporadici rilasciati mediante interrupt in un contesto di task periodici soft o non real-time
- Talvolta utilizzata in sistemi con carico real-time contenuto se il task aperiodico/sporadico ha una durata molto breve (-> *overhead*)
- E' possibile aggiungere lo *slack stealing*, ma nei sistemi priority-driven la sua realizzazione è complicata
- Esempi precedenti (sched. interrupt-driven + slack stealing):
 - se $E_A=0.8 \rightarrow T_{risp}=0.8$
 - se $E_A=2.1 \rightarrow T_{risp}=9.0$



Utilizzo di server per richieste aperiodiche

- Il tempo di risposta medio dei task aperiodici può essere migliorato, rispetto al servizio in background, *riservando un task periodico al servizio delle richieste aperiodiche*
- Denominato *Aperiodic server* o semplicemente *server*; oppure *Periodic server* (delle richieste aperiodiche)
- T_s = periodo del server, C_s = capacità del server (o budget)
- Il task *server* è schedulato con lo stesso algoritmo degli altri task periodici
- Il server $\tau_s(T_s, C_s)$ effettua il servizio delle richieste aperiodiche pendenti nei limiti della sua capacità C_s



Utilizzo di server per richieste aperiodiche

- L'ordine di servizio delle richieste aperiodiche pendenti è indipendente dall'algoritmo di scheduling dei task periodici
 - Ad es.: tempo di arrivo (FCFS), t. esec. (SJF), deadline (EDF)
- L'esecuzione dei job aperiodici non deve pregiudicare la schedulabilità dei task periodici. I task periodici tuttavia *possono essere ritardati*, a differenza del background
- Due classi: server adatti sia per schedulatori a priorità statica che a priorità dinamica (ad es. Polling Server, Deferrable Server) e server abbinati a schedulatori a priorità dinamica (ad es. Dynamic Priority Exchange)



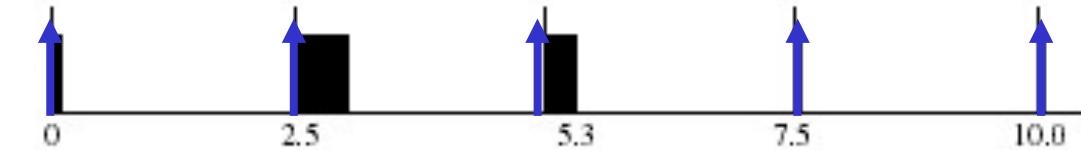
Polling server

- Il server $\tau_s(T_s, C_s)$ esegue le richieste aperiodiche pendenti nel suo istante di rilascio, e quelle che arrivano prima della sua conclusione, fino alla capacità C_s
- In assenza di richieste il server si sospende
- La capacità *non è preservata*, cioè non è disponibile per task aperiodici rilasciati successivamente entro il periodo del server
- Un job aperiodico *che arriva dopo che il server ha esaurito la coda del periodo corrente* sarà servito, se la capacità è sufficiente, nel periodo successivo
- La schedulazione dei task periodici, incluso il server, può essere sia RM che EDF



Esempio con polling server

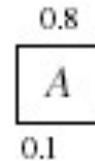
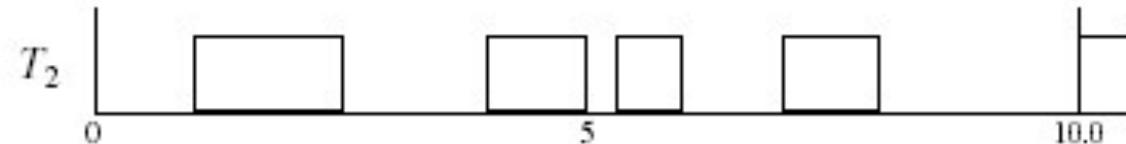
Server (poller)



$$T_1 = (3, 1)$$

$$T_2 = (10, 4)$$

$$T_s = (2.5, 0.5)$$

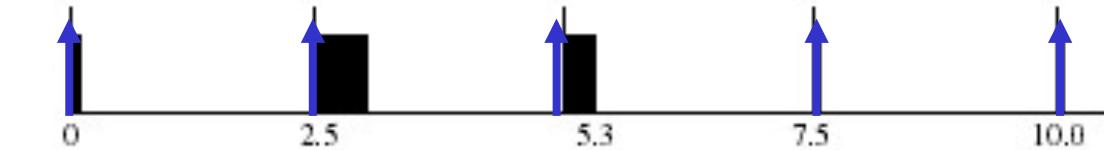


- Job aperiodico: $A = (R_A = 0.1, C_A = 0.8)$
- Con $T_s = (2.5, 0.5)$, $T_{\text{risp}} = 5.2$



Esempio con polling server

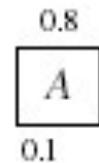
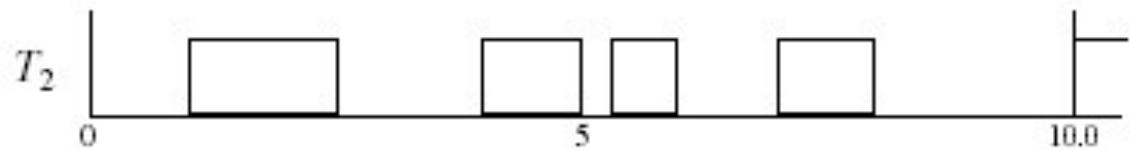
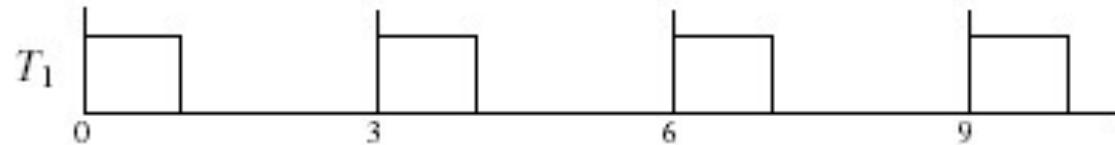
Server (poller)



$$T_1 = (3, 1)$$

$$T_2 = (10, 4)$$

$$T_s = (2.5, 0.5)$$



- Job aperiodico: $A = (R_A = 0.1, C_A = 0.8)$
- Con $T_s = (2.5, 0.5)$, $T_{\text{risp}} = 5.2$

Q: i task T_s , T_1 , T_2 sono tutti garantiti?



Polling Server

- Polling Server si comporta nel peggiore dei casi come un task periodico (T_s , C_s)
- Test di garanzia:

$$\sum_{i=1,n} C_i/T_i + C_s/T_s \leq U_{lub}$$

- U_{lub} dipende dall'algoritmo di scheduling considerato e dai parametri del polling server
- Ad esempio, con RM e utilizzando il bound di Liu e Layland:

$$\sum_{i=1,n} C_i/T_i + C_s/T_s \leq U_{LL}(n+1)$$



Polling Server in presenza di task sporadici

- In presenza di task sporadici da garantire per tutte le istanze future, per essi si deve fare riferimento alla deadline relativa
- Nel caso più generale sono presenti task periodici, sporadici ed aperiodici
- Test di garanzia con task sporadici (gestiti individualmente come i periodici) e task soft RT gestiti con Polling Server:

$$\sum_{i \text{ period}} C_i/T_i + \sum_{j \text{ sporad}} C_j/D_j + C_s/T_s \leq U_{lub}$$

- Eventuali task aperiodici di tipo hard devono essere garantiti valutando la *capacità totale di server* disponibile fino alle deadline specificate



Polling server per garantire un task sporadico

- Esercizio:
- Proporre le condizioni sufficienti per garantire *un singolo task sporadico* mediante *un polling server* con parametri (T_s, C_s)
- Sia $\tau_A(T_A, D_A, C_A)$ il task sporadico, in cui T_A è il tempo minimo di interarrivo, D_A la deadline relativa, C_A il tempo di esecuzione di caso peggiore
- «Sappiamo che τ_A può arrivare e vogliamo garantirci la possibilità di gestirlo rispettando la deadline»



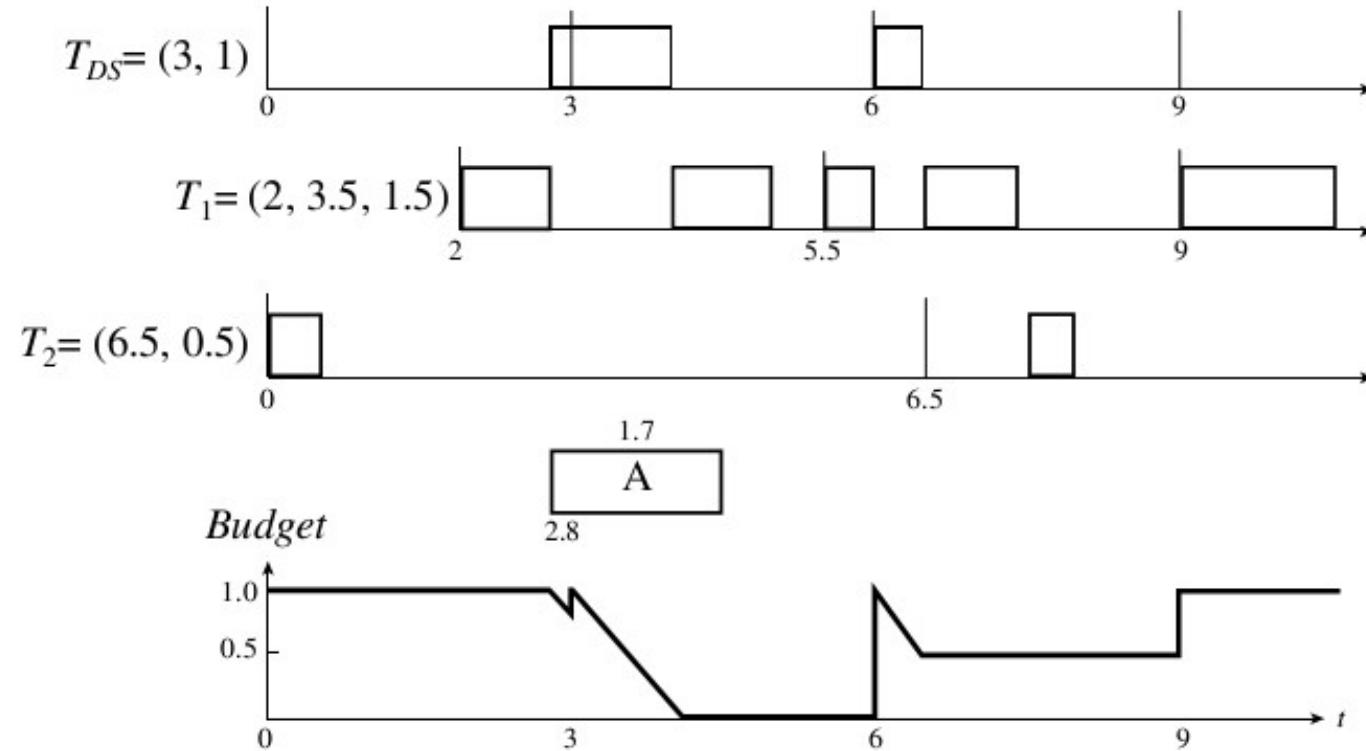
Deferrable Server

- Deferrable Server (DS): Server periodico, in genere ad alta priorità, dedicato al servizio di attività aperiodiche; utilizzato spesso con RM, utilizzabile anche con EDF
- DS *conserva* la sua capacità anche se, quando è rilasciato, non ci sono richieste pendenti → DS può fornire un servizio immediato alle richieste
- DS conserva tutta la capacità per tutta la durata del periodo T_s : le richieste possono essere servite in ogni istante con la priorità del server, purché la capacità non sia esaurita
- All'inizio di ogni periodo la capacità è ripristinata al valore nominale; quella inutilizzata del periodo precedente è persa



Deferrable Server

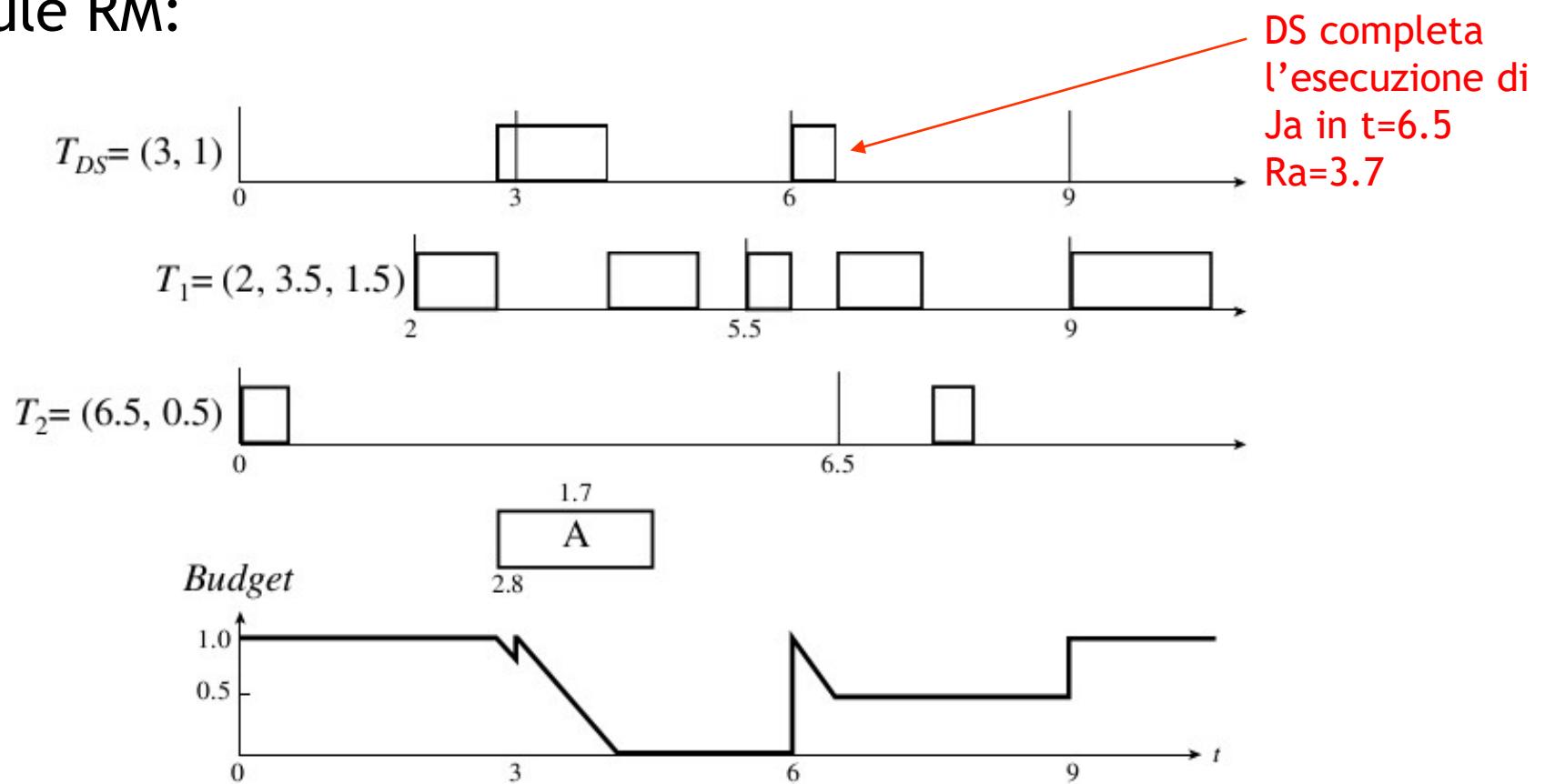
- Esempio: $\tau_{DS} = (3, 1)$, $\tau_1 = (\phi_1 = 2, 3.5, 1.5)$, $\tau_2 = (6.5, 0.5)$
- Schedule RM:





Deferrable Server

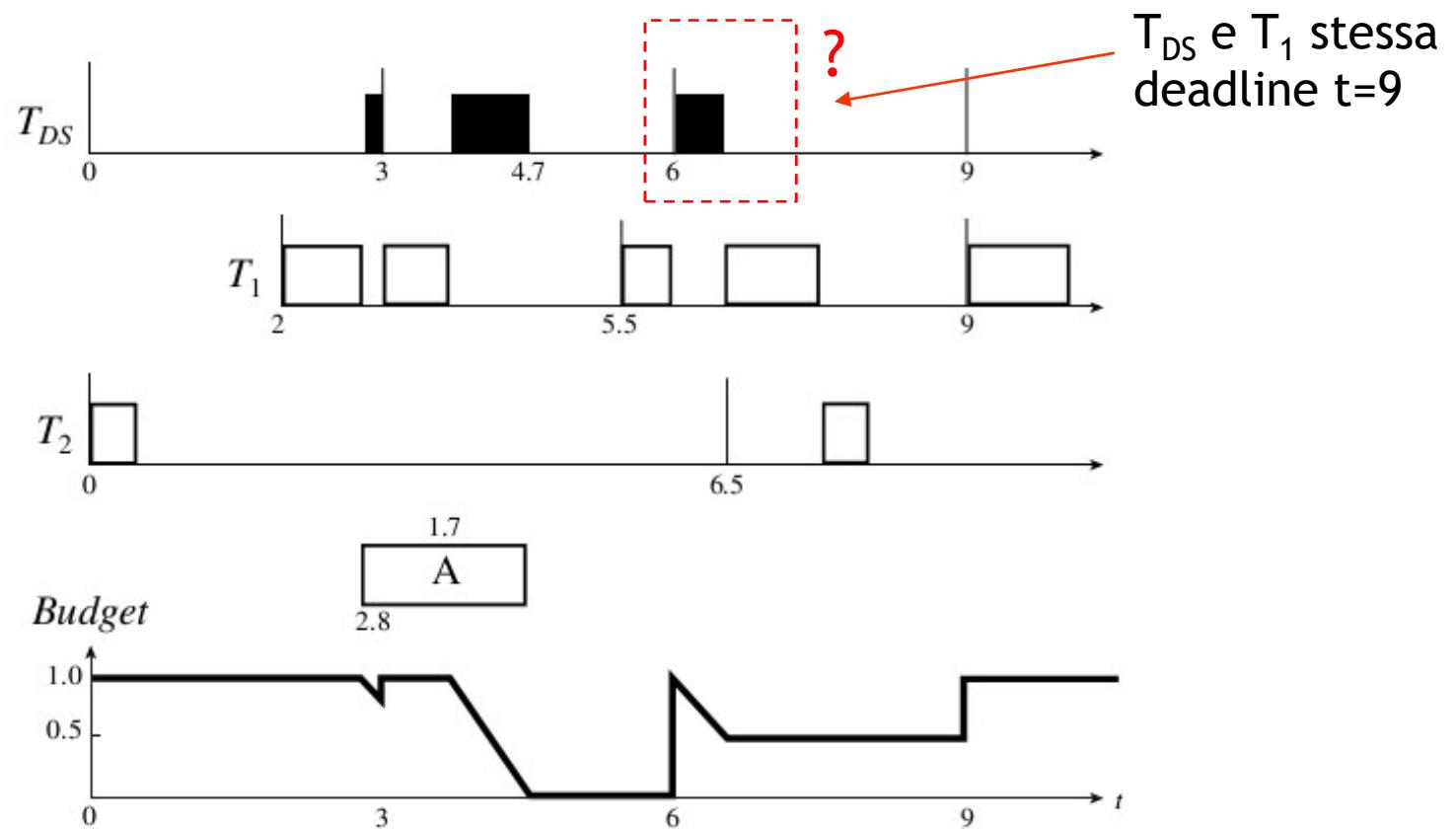
- Esempio: $\tau_{DS} = (3, 1)$, $\tau_1 = (\phi_1 = 2, 3.5, 1.5)$, $\tau_2 = (6.5, 0.5)$
- Schedule RM:





Deferrable Server

- Esempio: $\tau_{DS}(3,1)$, $\tau_1=(\phi_1=2,3.5,1.5)$, $\tau_2=(6.5,0.5)$
- Schedule EDF:





Deferrable Server

- Il tempo medio di risposta con DS è in genere nettamente migliore rispetto a PS
- DS tuttavia *non rispetta le ipotesi dei task periodici RM* (e in generale *work conserving*), in quanto il server è un task che non è sempre pronto all'inizio di ogni suo periodo
- La sua esecuzione può essere posticipata (deferred) o sospesa a causa della mancanza di richieste aperiodiche
- Il contributo di τ_s alla utilizzazione necessaria per garantire i task periodici è quindi superiore a C_s/T_s !

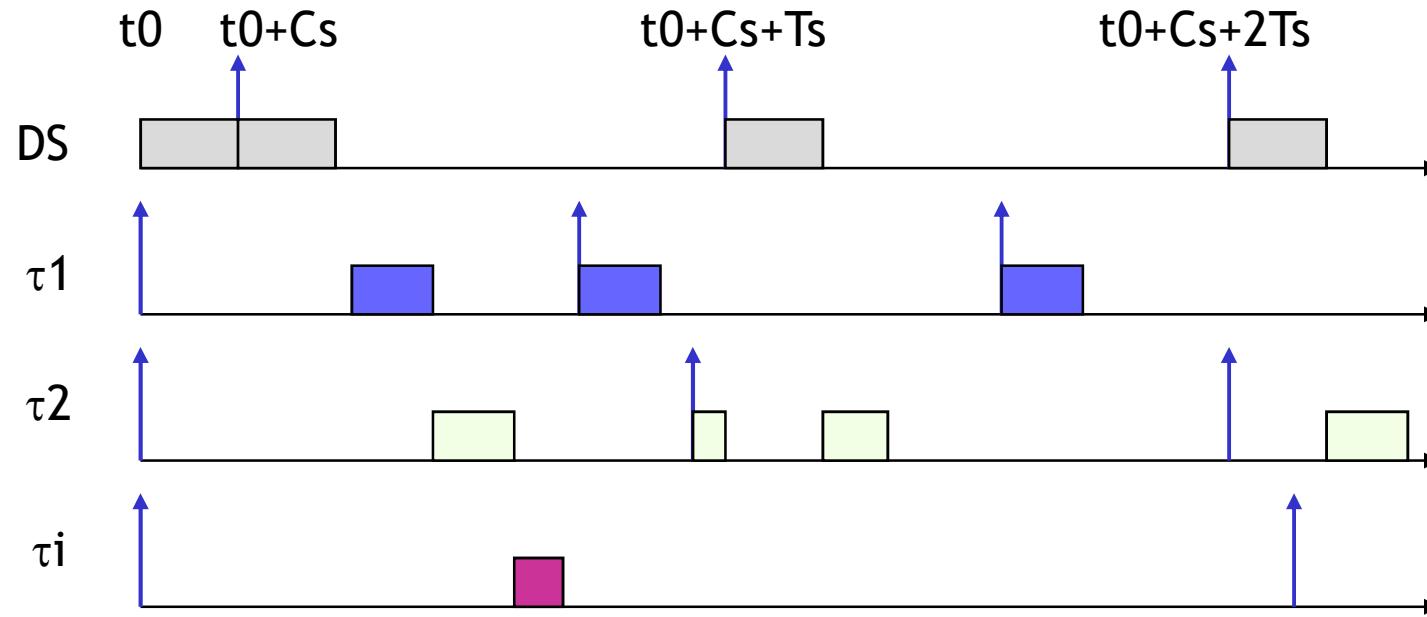


Deferrable Server

- In un sistema *a priorità fissa* in cui $D_i \leq T_i \forall i$ e in cui è presente un server DS (T_s, C_s) con *priorità massima* tra tutti i task, l'istante critico per un task periodico τ_i si verifica in un istante t_0 in cui:
 - è rilasciato un job $J_{i,c}$ di τ_i
 - in t_0 sono rilasciati job di tutti i task a priorità maggiore
 - il server ha ancora capacità C_s in t_0 , e in t_0 sono rilasciati job aperiodici tali da saturarlo
 - il successivo istante di ripristino della capacità del server è all'istante $t_0 + C_s$
- NB: Vale per tutti gli assegnamenti statici di priorità



Deferrable Server



- Intervallo critico in presenza di DS: temporizzazione che determina il massimo tempo di risposta per τ_i



Deferrable Server

- L'analisi dell'intervallo critico evidenzia come *DS determini un carico maggiore* rispetto ad un task periodico di pari periodo e con tempo di esecuzione uguale alla capacità del server
- Per n task periodici schedulati in modo RM assieme ad un DS(T_s, C_s) con *periodo Ts arbitrario* e priorità RM, è possibile dare conto del *tempo di blocco aggiuntivo* che subisce *un task i a priorità inferiore* a quella del server considerando per esso $U_i = (C_i + B_i) / T_i = (C_i + C_s) / T_i$
- → condizione sufficiente per schedulabilità di τ_i :

$$\sum_{j=1,i} C_j / T_j + C_s / T_s + C_s / T_i \leq U_{RM} (i+1)$$



Deferrable Server

- Condizione sufficiente per schedulabilità di un task generico τ_i in presenza di un deferrable server DS e scheduling RM (DS con priorità RM) con priorità superiore a τ_i :

$$\sum_{j=1,i} C_j/T_j + C_s/T_s + C_s/T_i \leq U_{RM}(i+1)$$

- Osserviamo che:
 - il blocco dovuto al DS si aggiunge ad un eventuale blocco per sezioni non revocabili a priorità inferiore
 - i task con priorità superiore a DS non sono influenzati
 - sono presenti $n+1$ task
 - occorre verificare se anche DS è garantito
 - il test va eseguito task per task, non è un bound in forma chiusa
 - il bound RM fornisce una condizione in generale solo sufficiente



Deferrable Server

- Esiste un bound in forma chiusa per il caso RM integrato da un DS solo sotto specifiche ipotesi sui valori dei periodi dei task periodici in relazione a quello del server e se il DS è il task a rate massimo e massima priorità
- In generale, conviene aggiungere al DS un BS (background server) che possa proseguire l'elaborazione dei job aperiodici, in presenza di slack time, anche dopo che DS ha esaurito la capacità
- Mediante l'analisi time-demand, in un sistema a priorità statica il DS può essere utilizzato e garantito anche sotto ipotesi più ampie, in particolare nel caso in cui il server abbia priorità arbitraria



Deferrable Server con EDF

- Dati n task periodici indipendenti e revocabili, in esecuzione con scheduling EDF assieme ad un DS(T_s, C_s) con periodo T_s arbitrario, ciascun task periodico τ_i rispetta le proprie deadline se:

$$\sum_{j=1,n} C_j / \min(D_j, T_j) + C_s / T_s + C_s / T_s \cdot ((T_s - C_s) / D_i) \leq 1$$

- Nel caso $D_j = T_j$, la condizione di garanzia si può riscrivere come:

$$\sum_{j=1,n} C_j / T_j + C_s / T_s + C_s / T_i \cdot (1 - U_s) \leq 1$$

- In pratica, nel caso EDF il tempo di blocco causato da DS subito da *ciascun task* non è $B_i = C_s$ ma è $B_i = C_s (1 - U_s)$



Deferrable Server con EDF

- Con EDF il tempo di blocco causato da DS è subito *da ciascun task* non è $B_i = C_s$ ma è $B_i = C_s (1 - U_s)$, cioè è un po' inferiore rispetto ad RM
- Tuttavia il blocco può essere subito da *tutti i task*, mentre nel caso RM solo i task a priorità inferiore di DS sono influenzati
- Poiché il DS preserva la banda fino alla fine del suo periodo, le ipotesi del teorema di Baker sono violate, ed il DS può ritardare qualsiasi task
- In presenza di più deferrable server con EDF è possibile considerare ciascuno di essi in modo additivo, ripetendo gli ultimi due termini della formula



Sporadic server

- Nella versione più semplice, lo Sporadic Server viene utilizzato assieme ad RM
- La capacità viene ripristinata non all'inizio di ogni periodo ma solo dopo che è stata consumata, parzialmente o totalmente, da una richiesta aperiodica
- Il server dal punto di vista computazionale si comporta esattamente come un task di pari periodo e capacità
- Vale pertanto la seguente formula di garanzia:

$$\sum_{i=1,n} C_i/T_i + C_s/T_s \leq U_{lub}$$



Altri server aperiodici

- Esistono molti altri tipi di server, alcuni dei quali abbinabili solo a schedulazione di tipo deadline-driven
 - *Server a priorità dinamica*
- Il *tradeoff* è tra la capacità del server di garantire un maggior numero di task periodici e aperiodici e la complessità realizzativa, ovvero l'overhead del server



Ripasso: test di garanzia per PS e DS

- Polling Server con RM:

$$\sum_{i=1,n} C_i/T_i + C_s/T_s \leq U_{RM}(n+1)$$

- Polling Server con EDF:

$$\sum_{i=1,n} C_i/T_i + C_s/T_s \leq 1$$

- Deferrable Server con RM:

$$\sum_{j=1,i} C_j/T_j + C_s/T_s + C_s/T_i \leq U_{RM} (i+1)$$

per ogni τ_i con priorità inferiore a τ_{DS}

- Deferrable Server con EDF:

$$\sum_{j=1,n} C_j/T_j + C_s/T_s + C_s/T_i \cdot (1 - U_s) \leq 1 \text{ per ogni } \tau_i$$



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Protocolli di accesso a risorse condivise per task in tempo reale

prof. Stefano Caselli



Dove sono finite le risorse?

- La contesa per le risorse influenza il comportamento e la schedulabilità dei job
 - L'accesso non regolato alle risorse produce errori inaccettabili e deve essere compatibile con i vincoli di tempo reale
- Sono stati sviluppati protocolli di accesso alle risorse che attenuano gli effetti della contesa e definiti metodi per tenerne conto dal punto di vista delle garanzie real-time
- Nei sistemi clock-driven l'accesso alle risorse è garantito all'interno di slice, e quindi è pianificato a priori
- → problema rilevante per i sistemi priority-driven !



Modello delle risorse

-
- Unità di risorse riusabili in modo sequenziale
 - Richiedono accesso esclusivo per l'utilizzo; una volta assegnate ad un job possono essere utilizzate da altri job solo dopo che sono state rilasciate
 - Presuppongono controllo dell'accesso mediante primitive del SO: ad es. *lock(x)* e *unlock(x)* oppure *wait(sem)* e *signal(sem)*



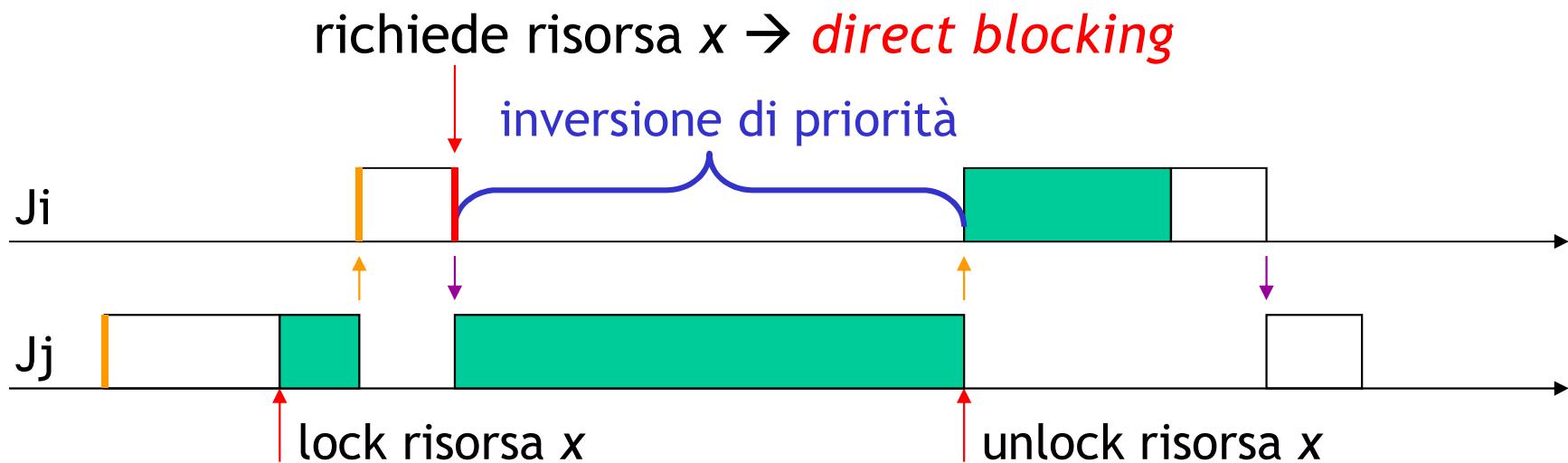
Inversione di priorità

- In presenza di sincronizzazioni si può verificare una situazione di *inversione della priorità*, in cui un job a priorità inferiore ne *blocca* uno a priorità più elevata (ad esempio su un semaforo mutex)



Inversione di priorità

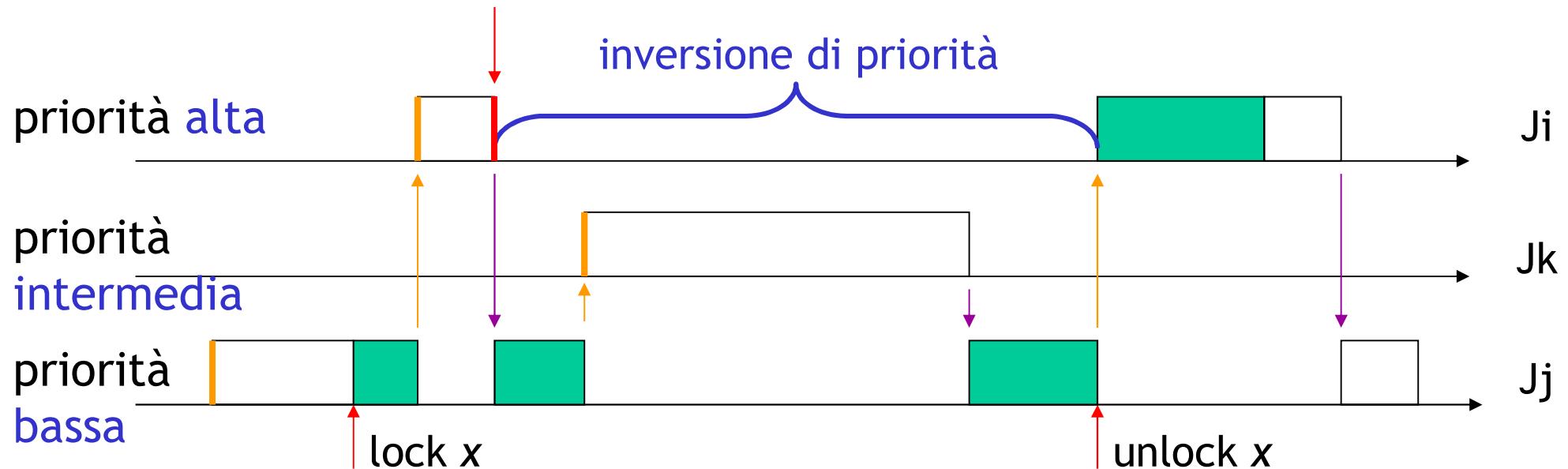
- Un job ad alta priorità è ritardato nell'esecuzione da parte di un job a priorità più bassa
 - → la relazione di priorità è invertita!
- J_i job ad alta priorità; J_j job a bassa priorità:





Inversione di priorità per un tempo illimitato

- La durata della inversione di priorità non è funzione solamente del tempo necessario al job a bassa priorità per eseguire la sezione critica!
- *Unbounded priority inversion*





Tempo di risposta di un job

- Nel caso peggiore è determinato da:
 - Tempo di esecuzione del job
 - Preemption subita da altri job a priorità maggiore
 - Tempo di blocco: ritardo subito nello stato bloccato
- Nel caso più favorevole il tempo di blocco è una funzione semplice dei ritardi subiti mentre altri job sono in sezione critica
- Se così non è, il tempo di blocco è difficile da calcolare



Inversione di priorità

- In presenza di sincronizzazioni si può verificare una situazione di *inversione della priorità*, in cui un job a priorità inferiore ne *blocca* uno a priorità più elevata (ad esempio su un semaforo mutex)
- Il job a priorità inferiore può a sua volta essere interrotto da uno a priorità intermedia
- ⇒ Occorre *innalzare temporaneamente la priorità di un job* a quella più alta tra tutti i job che esso blocca (*priority boosting*)

Tecniche per evitare inversione di priorità incontrollata



- Sezioni critiche non revocabili
- Protocollo Priority Inheritance
- Protocollo Priority Ceiling



Sezioni critiche non revocabili

- L'approccio è denominato protocollo *NPCS* (*Non-Preemptive Critical Sections*) (Mok, 1983)
- Caratteristiche:
 1. Quando un job *richiede* una risorsa, essa è libera per definizione e la risorsa gli viene allocata
 2. Quando un job *detiene* una risorsa, esso esegue ad una priorità superiore a tutti gli altri job e *non può subire preemption*
- Nota: con NPCS non si può verificare deadlock



Sezioni critiche non revocabili

- Teorema: Con il protocollo NPCS un job J_i può essere bloccato una sola volta
- In un sistema RT *a priorità fissa* con n task, il tempo di blocco massimo B_i che un task periodico τ_i può subire a causa della contesa sulle risorse è pari alla *più lunga sezione critica* di *tutti* i task a priorità inferiore
- Sia ξ_k la più lunga sezione critica del task τ_k
- Ordinando i task con priorità decrescente, si ha:

$$B_i = \max_{i+1 \leq k \leq n} \xi_k$$



Sezioni critiche non revocabili

- Con EDF, il job di un task τ_i con deadline relativa D_i può subire blocco solo da job J_k di task con deadline relativa $D_k > D_i$
- Sia ξ_k la più lunga sezione critica del task τ_k
- Il massimo tempo di blocco B_i che un task periodico τ_i può subire a causa della contesa sulle risorse è ancora pari alla più lunga sezione critica di tutti i task con deadline relativa maggiore:

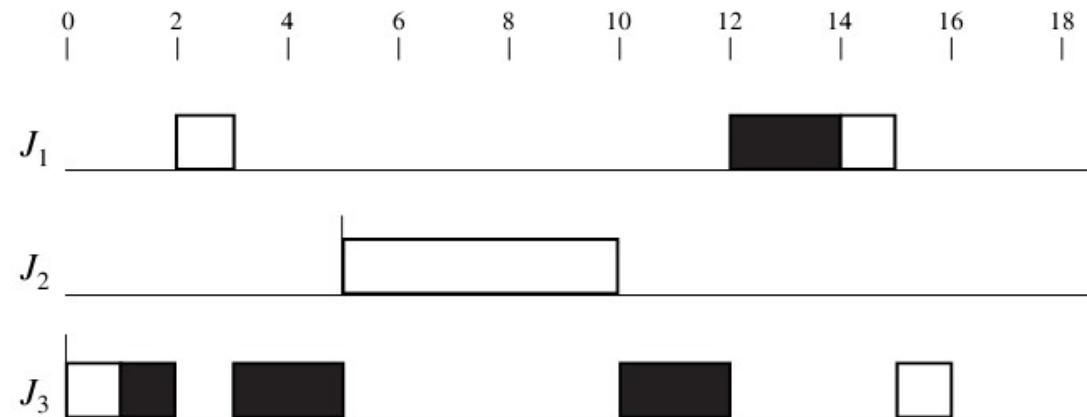
$$B_i = \max_{i+1 \leq k \leq n} \xi_k$$

- ove però i task sono ordinati in base alle deadline relative crescenti ($i < j$ se $D_i < D_j$)

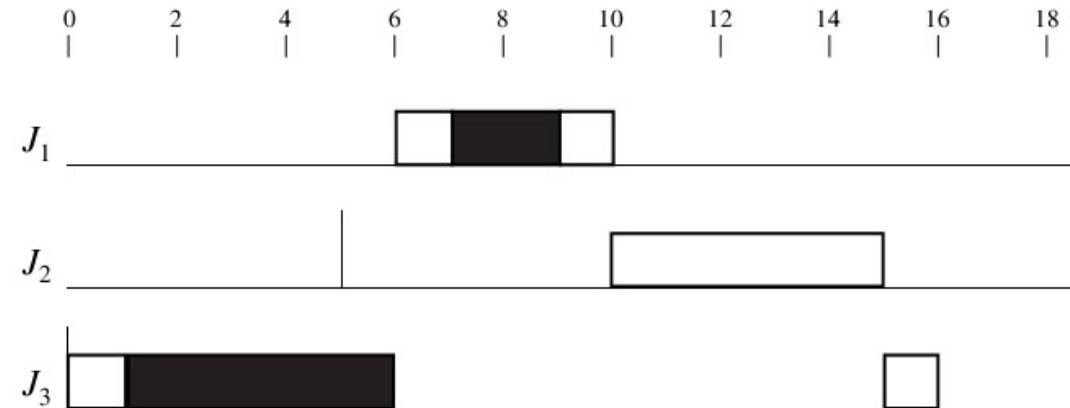


Esempio con sezioni critiche non revocabili

- Situazione di inversione di priorità:



- Con NPCS:





Sezioni critiche non revocabili

- Caratteristiche del protocollo NPCS:
- Semplice, anche con numero arbitrario di risorse
- Trasparente al programmatore! (ci pensa il SO/supporto run-time)
- Funziona bene se *le sezioni critiche sono tutte brevi*, anche nel caso di sezioni critiche numerose e conflitto tra molti task
- Svantaggio: un job può essere bloccato da *un qualunque job a priorità inferiore* anche se non c'è alcuna contesa/ conflitto tra loro
- Grave se le sezioni critiche non sono tutte brevi!



Protocollo priority inheritance

- [Sha et al., 1990]
- Protocollo applicabile con *qualsiasi algoritmo priority-driven*, con priorità statiche o dinamiche, *preemptive* (→ EDF, RM, DM)
- Trasparente al programmatore
- Previene inversioni di priorità di durata incontrollata
- Realizzazione relativamente semplice
- Non risolve tutti i problemi:
 - Possibili catene di attesa lunghe, nel caso peggiore
 - Non previene deadlock, quindi attese indefinite



Protocollo priority inheritance: ipotesi

- Priorità nominale assegnata ai job in base al criterio di priorità adottato dall'algoritmo
- FCFS tra job di eguale priorità
- $J_1, J_2, \dots J_n$ con priorità nominali decrescenti
- Sezioni critiche anche annidate, con annidamento regolare
- Accesso a sezioni critiche con semafori binari o primitive lock/unlock:
 $lock(Ris) \leftarrow---- sez critica ----- \rightarrow unlock(Ris)$



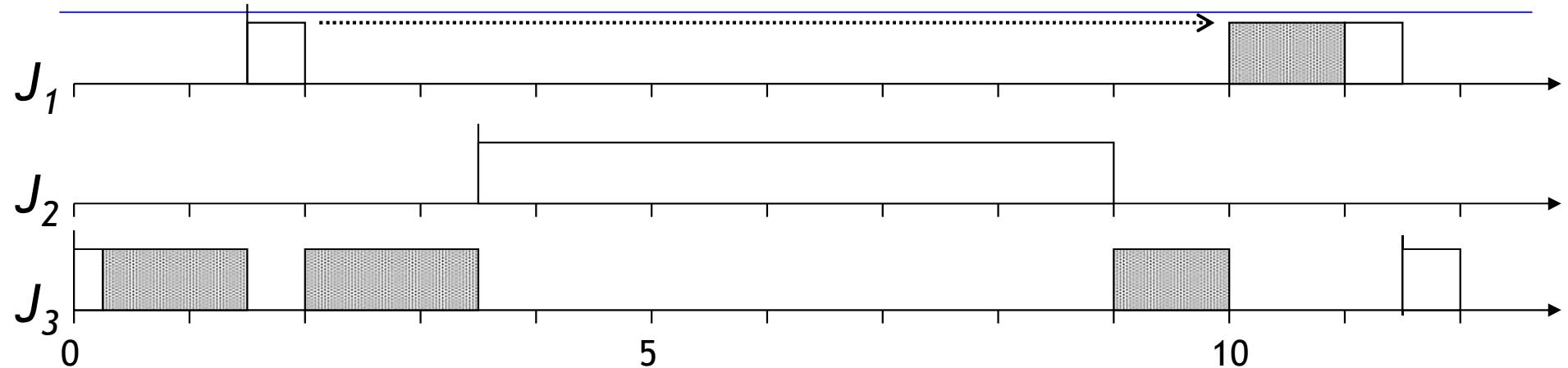
Protocollo priority inheritance

- Se un job J_a si blocca all'ingresso di una sezione critica esso *trasmette la propria priorità* a J_b che la detiene
- J_b *esegue alla priorità massima* tra quelle dei job che sta bloccando a causa delle risorse che detiene
- All'uscita dalla sezione critica il job J_b è riportato alla sua priorità naturale
- L'ereditarietà della priorità è transitiva
- Due situazioni di blocco per un job ad alta priorità:
 - Perché la sezione critica a cui vuole accedere è occupata da un job a priorità inferiore → *blocco diretto*
 - Perché un job a bassa priorità ha ereditato una priorità maggiore in sezione critica → *blocco push-through*



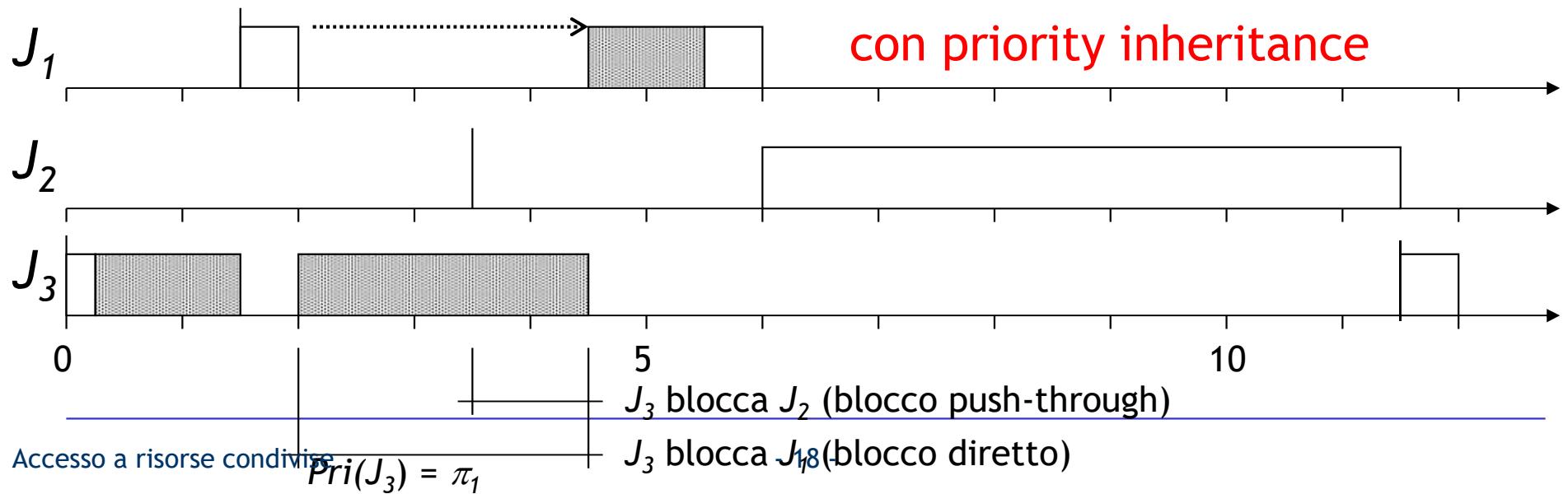
Priority Inheritance

senza priority inheritance



$$\pi_1 > \pi_2 > \pi_3$$

con priority inheritance



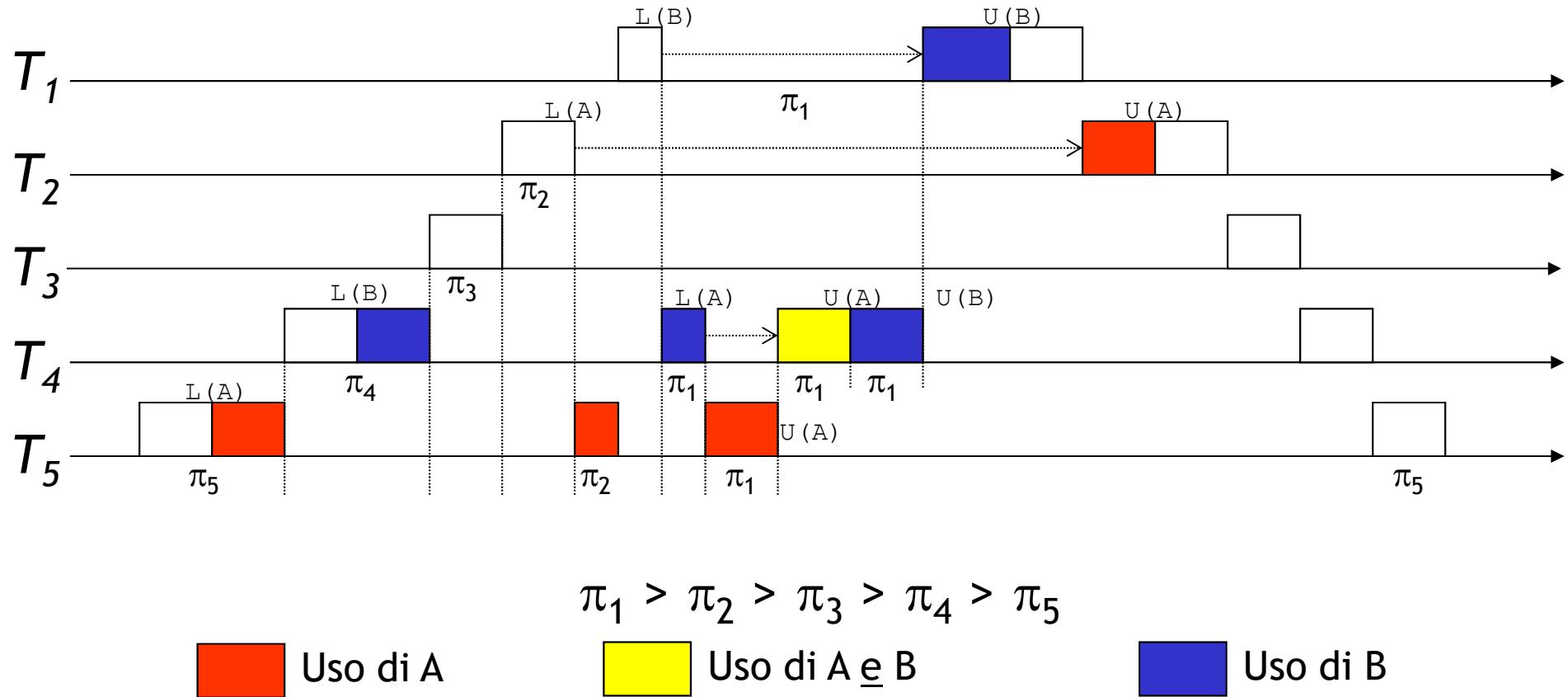


Priority Inheritance

- Teoremi:
- T1: Un job J_k può essere bloccato da un job a priorità inferiore J_l *una sola volta*
→ Nel caso peggiore J_k può essere bloccato da ciascun job a priorità inferiore su una *sezione critica diversa*
- T2: Se ci sono m lock o semafori su cui J_k può dover attendere *a causa di blocco diretto o indiretto*, J_k può essere bloccato al più m volte
- T3: Un job J_k può essere bloccato al più per la durata di $\min(n,m)$ sezioni critiche, dove n è il numero di job a priorità inferiore di J_k ed m è il numero di lock o semafori su cui J_k può attendere *per blocco diretto o indiretto*



Protocollo Priority Inheritance (PIP) - Esempio



$$\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$$

█ Uso di A

█ Uso di A e B

█ Uso di B

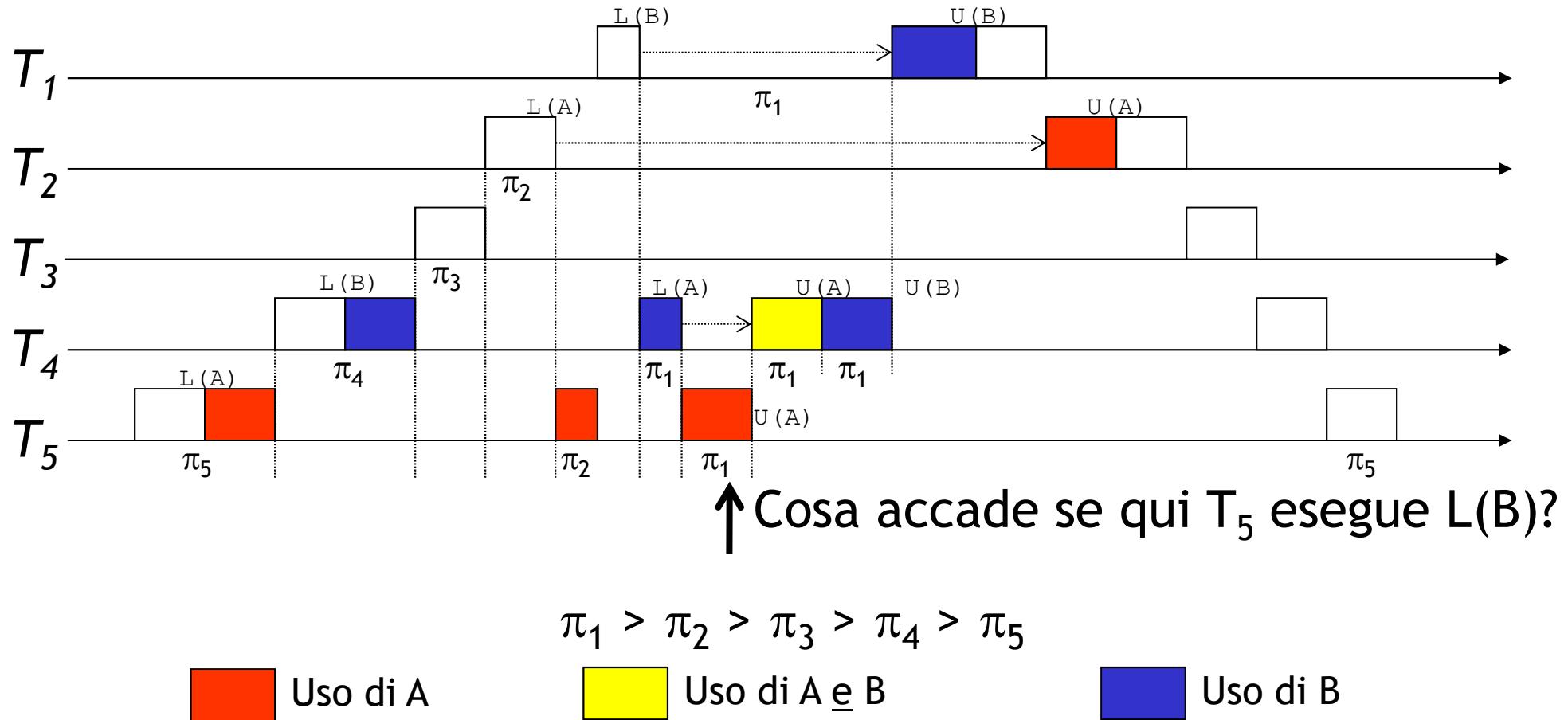


Protocollo PIP: Garanzie

- E' possibile integrare l'effetto della contesa nei bound che caratterizzano l'algoritmo priority-driven considerando il *tempo di blocco massimo* B_i che ciascun task può subire
- RM: $\forall i, 1 \leq i \leq n \quad \sum_{k=1,i} C_k/T_k + B_i/T_i \leq U_{lub}(i)$
- EDF: $\forall i, 1 \leq i \leq n \quad \sum_{k=1,n} C_k/T_k + B_i/T_i \leq 1$
- Occorre calcolare i B_i sulla base dei teoremi precedenti e procedere valutando le garanzie per ciascun task, con costo $O(n)$



Protocollo Priority Inheritance (PIP) - Problema





Inversione di priorità e garanzia

- Dati un algoritmo di scheduling priority driven di task periodici ed un protocollo per il controllo della inversione di priorità, come si applicano i test di garanzia?
- Con algoritmi statici (RM, DM, altro): un job può essere bloccato solamente da job appartenenti a task a priorità inferiore
- Con EDF vale il teorema di Baker: un job può essere bloccato solamente da job appartenenti a task con deadline relativa maggiore
- Con algoritmi dinamici diversi da EDF: un job può essere bloccato da job appartenenti a tutti gli altri task



Esercizi su task interagenti

- Ci limitiamo qui a casi privi di annidamenti di sezioni critiche
 - Semplificazione!
 - Non si può verificare deadlock: manca possesso e attesa su risorse condivise
- Es. 1: Determinare la schedulabilità RM dei seguenti task (Bi è il tempo di blocco):

	Ci	Ti	Bi
J1	1	2	1
J2	1	4	1
J3	2	8	0



Esercizi su task interagenti

- Es. 2: Blocking time computation con PIP e NPCS
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3
J1	1	2	0
J2	0	9	3
J3	8	7	0
J4	6	5	4

- Determinare il *massimo tempo di blocco* e il *numero di blocchi* subiti da ciascun task ($\pi_1 > \pi_2 > \pi_3 > \pi_4$)



Esercizi su task interagenti

- Es. 2: Blocking time computation con PIP e NPCS
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3	Ni Bi	Ni Bi	Ni Bi
J1	1	2	0	2 17	1 9	1 9
J2	0	9	3	2 13	1 8	1 8
J3	8	7	0	1 6	1 6	1 9
J4	6	5	4	0 0	0 0	1 9

PIP **NPCS** **NPCS non EDF**

- Massimo tempo di blocco Bi e numero di blocchi subiti Ni da ciascun task ($\pi_1 > \pi_2 > \pi_3 > \pi_4$ per RM e EDF/Baker)



Esercizi su task interagenti

- Es. 2bis: Blocking time computation con PIP e NPCS
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3	Ni Bi	Ni Bi	Ni Bi
J1	1	2	0	2 17	1 41	1 41
J2	0	9	3	2 49	1 41	1 41
J3	8	7	0	1 41	1 41	1 41
J4	6	5	41	0 0	0 0	1 9

PIP NPCS NPCS non EDF

- $\pi_1 > \pi_2 > \pi_3 > \pi_4$ per RM e EDF/Baker



Esercizi PIP e NPCS

- Es. 3: Blocking time computation
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3	C4	
J1	1		3;2		// J1 accede 2 volte a C3
J2		1	1;2	3	// per J2 caso peggiore =2
J3	2			1	
J4	1	1		4	

- Determinare il massimo tempo di blocco e il numero di blocchi subiti da ciascun task sia con NPCS che con PIP



Esercizi PIP e NPCS

- Es. 4: Blocking time computation
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3	C4
J1	1		3;2	
J2		1	1;2	
J3	1			80
J4	1	2		100

- Determinare il massimo tempo di blocco e il numero di blocchi subiti da ciascun task sia con NPCS che con PIP



Esercizi PIP e NPCS

- Es. 4: Blocking time computation
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3	C4
J1	1		3;2	
J2		1	1;2	
J3	1			80
J4	1	2		100

Bi	Ni
3	2
3	2
100	1
0	0

Bi	Ni
100	1
100	1
100	1
0	0

Perché J_1 e J_2
sono ritardati da
 J_3 e J_4 , lenti?
⇒ NPCS poco
gradito ...

- Determinare il massimo tempo di blocco e il numero di blocchi subiti da ciascun task sia con NPCS che con PIP



Esercizi PIP e NPCS

- Es. 5: Blocking time computation
- Dati i task periodici $\{J_i\}$, che competono per le sezioni critiche $\{C_j\}$ secondo la seguente tabella:

	C1	C2	C3
J1	1	0	0
J2	0	20	30
J3	0	0	0
J4	4	2	2
J5	2	3	3

- Determinare il massimo tempo di blocco e il numero di blocchi subiti da ciascun task sia con NPCS che con PIP



Protocollo priority ceiling

- Viene attribuito un valore di *ceiling di priorità* pari alla massima priorità dei task che possono accedere alla risorsa
- Un job che accede ad una risorsa acquisisce una priorità pari al ceiling
- Nella analisi di schedulabilità occorre ancora considerare il tempo di blocco massimo che ciascun task può subire
- Il protocollo Priority Ceiling *previene il deadlock*; inoltre ciascun job può essere *bloccato una sola volta*
- PCP si presta ad essere integrato con algoritmi a *priorità statica*, mentre l'integrazione con algoritmi a priorità dinamica è complicata (il ceiling delle risorse cambia!)





Task interagenti nei sistemi real-time

- Test O(1) per NPCS, PIP e PCP:

$$\sum_{i=1,n} \underline{C_i/T_i} + \max_{i=1,n-1} (B_i/T_i) \leq U_{lub}(n)$$

- (i tempi di blocco B_i cambiano tra NPCS, PIP, PCP)
- Tutto assieme, più semplice ma più restrittivo
- Alcuni «detti» comuni della programmazione RT:
 - The best lock is the one that you don't need!
 - Don't share if you can avoid it
 - Make sharing tasks at the same priority if you can, or fuse them in single task



Posizione un po' estrema, anche se autorevole

- Philip Koopman (CMU), nel libro «*Better embedded system software*», pag 147:
 - Don't use EDF scheduling:
 - (with EDF or MLF) system behavior is very bad if system load goes above 100%, whereas RMS is better behaved in overload conditions
 - Moreover, EDF and MLF require run-time changes to priorities, which is more complex than the fixed priority approach used by RMS
 - --> *Stay away from EDF and Least Laxity, use RMS if you can*
- Voi siete gli esperti che possono utilizzare in modo competente anche EDF!

But what really happened on Mars, in July 1997?



Sojourner incontra la roccia Yogi



Il rover visto da Pathfinder



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Analisi di schedulabilità



Analisi di schedulabilità

- Approcci:
 - Utilizzazione schedulabile dell'algoritmo
 - con EDF $U_{tot} \leq 1$; con RM: LL, KM, HB, altri
 - Analisi del tempo di risposta ovvero del tempo di processore richiesto:
 - Metodo strutturato per effettuare analisi da istante critico o con relazioni di fase definite
 - Anche in varianti algebriche / computazionali
 - Altri metodi:
 - simulazione, dry run, hardware-in-the-loop, finger-crossed method



Analisi del tempo di risposta (Audsley et al., '90)

- Denominata anche *time-demand analysis*, realizza un *test di schedulabilità* per *task con priorità fissa*
- Si applica ad *algoritmi a priorità fissa arbitrari*; di particolare utilità quando manca una stima ragionevole della utilizzazione schedutabile (ad es., DM)
- Si basa sull'idea di generare la situazione di caso peggiore, l'*istante critico*, e di simulare gli eventi fino alla prima deadline di ogni task
- Dati di ingresso: $\{\tau_i\} = \{(T_i, C_i, D_i)\}$
- L'algoritmo di Audsley verifica un task alla volta per determinare se i *tempi di risposta* dei job, calcolati a partire da (T_i, C_i) rispettano le deadline relative D_i



Analisi del tempo di risposta

- Ipotesi: $D_i \leq T_i$, $U \leq 1$
(per estensione al caso di deadline relative arbitrarie, v. Liu)
- Osservazione: ogni task è influenzato solo dai task a più alta priorità, da cui subisce revocate
- Si inizia dal task a priorità più elevata e si procede per priorità decrescente
- $R_i = \text{massimo tempo di risposta di } \tau_i$; si ottiene a partire dall'istante critico t_0 . τ_i è schedulabile se:
$$R_i = C_i + I_i \leq D_i \quad i=1,2, \dots, N$$
- I_i è l'*interferenza* sul tempo di risposta di τ_i da parte di job a più alta priorità dovuta a preemption



Analisi del tempo di risposta

- In generale, in un istante t_0+t la *richiesta totale di tempo di processore* (time-demand) $W_i(t)$ di un job di τ_i e di tutti gli altri job a più alta priorità rilasciati in $[t_0, t_0+t]$ è:

$$W_i(t) = C_i + \sum_{k=1, i-1} \lceil t/T_k \rceil C_k \quad \text{per } 0 < t \leq T_i$$

- Il job di τ_i può rispettare la sua deadline t_0+D_i , se in un qualche istante t_0+t entro la deadline la *disponibilità di tempo di processore*, t , soddisfa la *richiesta totale* $W_i(t)$:

$$\exists t \mid W_i(t) \leq t \quad \text{per qualche } t \leq D_i$$



Analisi del tempo di risposta

- Se $W_i(t) > t$ per ogni $t \leq D_i$
il job *non può* completare entro la deadline, nel caso venga rilasciato nell'istante critico
- L'insieme di task non è garantito in base al test di Audsley
- L'insieme di task *potrebbe* essere garantito solo se è possibile controllare la fase di rilascio ed il jitter dei task in modo da evitare situazioni di rilascio all'istante critico
- Se il test di Audsley non è soddisfatto, per garantire i task occorre simularne l'evoluzione, con le esatte fasi di rilascio previste, per un iperperiodo (assumendo assenza di jitter)



Time-demand analysis

- La tecnica è denominata *time-demand analysis*, analisi della funzione di *richiesta del tempo di processore*
- La funzione di time-demand $W_i(t)$ del task τ_i è:
$$W_i(t) = C_i + \sum_{k=1, i-1} \lceil t/T_k \rceil C_k \quad \text{per } 0 < t \leq T_i$$
- Per qualunque assegnazione statica di priorità, per task periodici con deadline $D_i \leq T_i$ è possibile eseguire l'analisi del tempo richiesto per via grafica, a partire dall'istante critico

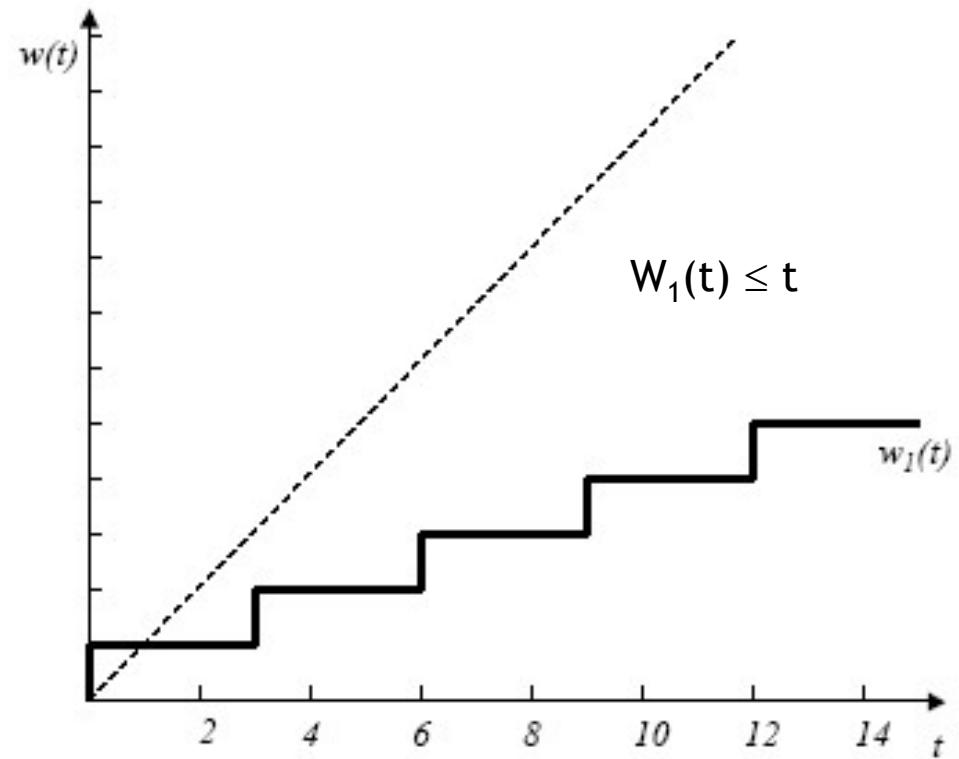


Analisi del tempo di risposta

- Grafico del tempo richiesto al processore dal task τ_i :

Ad es., per $\tau_1 = (3, 1)$,
a massima priorità:

L'ordinata rappresenta
la *richiesta* di
processore, la retta a
pendenza 1 la
disponibilità di tempo





Analisi del tempo di risposta - Esempio

$$\tau_1 = (3, 1) \quad U_1 = 0.333$$

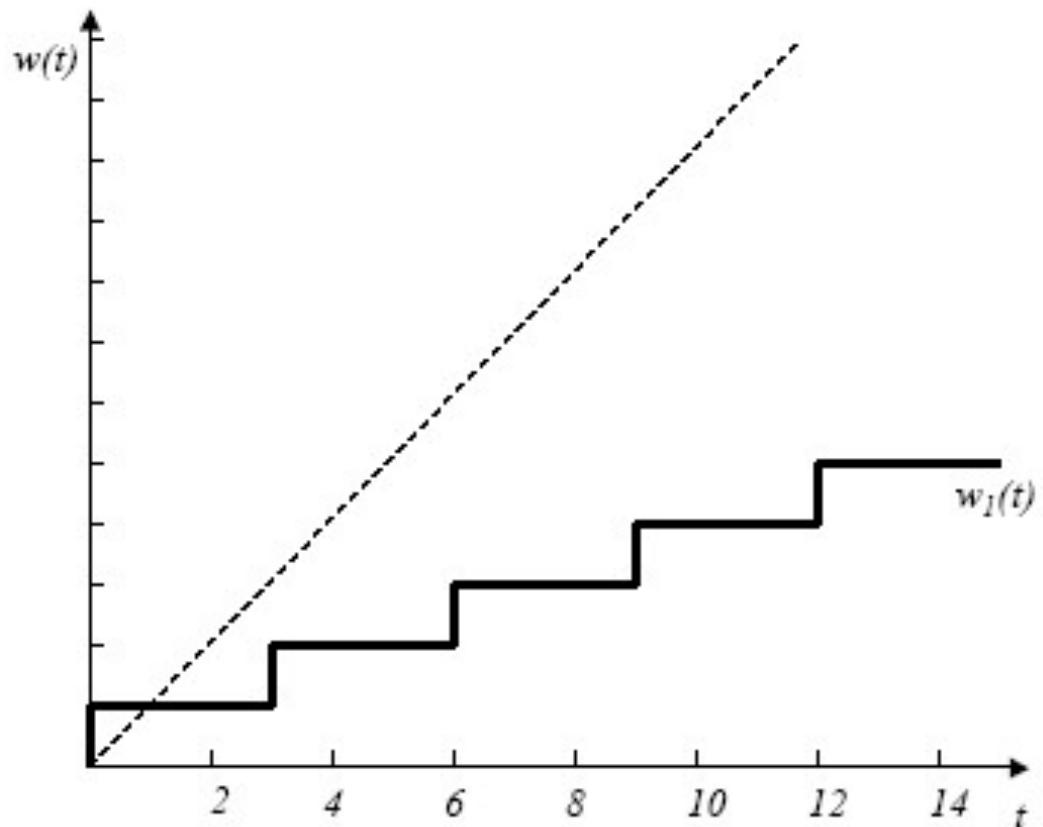
$$\tau_2 = (5, 1.5) \quad U_{12} = 0.633$$

$$\tau_3 = (7, 1.25) \quad U_{123} = 0.812$$

$$\tau_4 = (8, 0.5) \quad U_{1234} = 0.874$$

In base a $U_{LL}(RM)$ e HB,
solo τ_1 e τ_2 sono garantiti

τ_1 completa in $t=1$ e
rispetta la deadline
(è a max priorità)





Analisi del tempo di risposta - Esempio

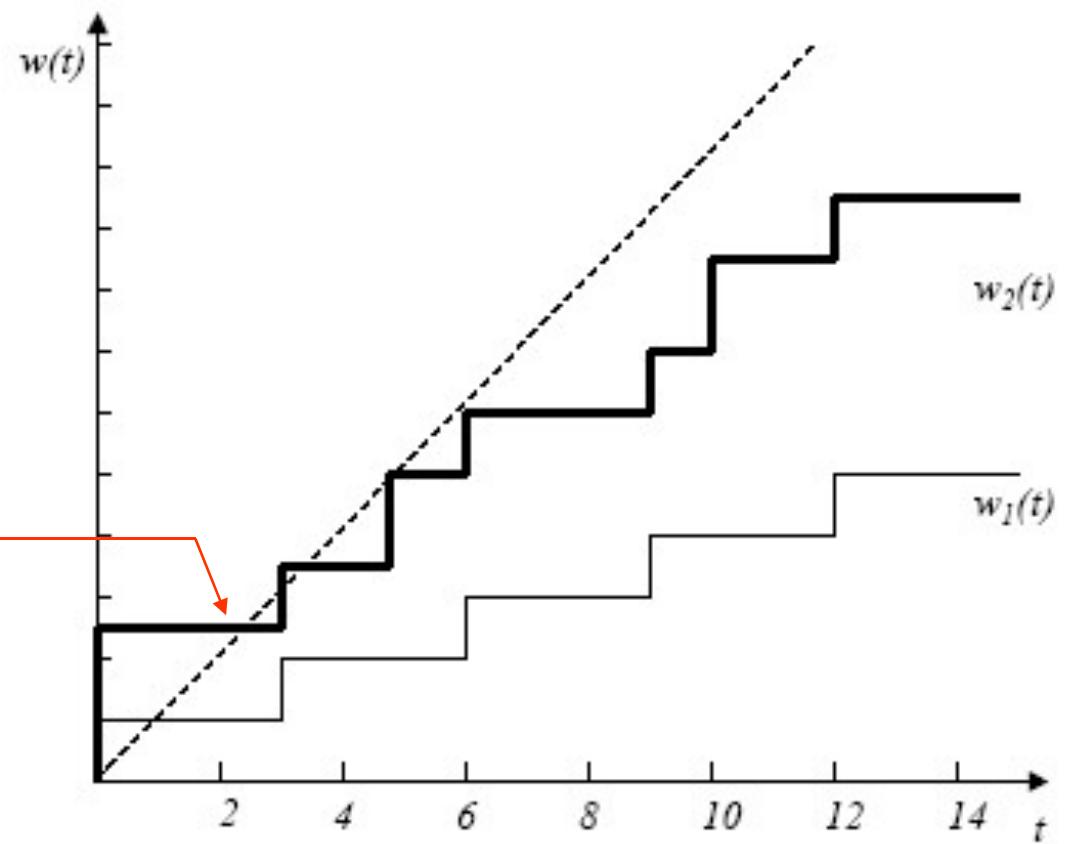
$$\tau_1 = (3, 1)$$

$$\tau_2 = (5, 1.5)$$

$$\tau_3 = (7, 1.25)$$

$$\tau_4 = (8, 0.5)$$

- All'istante 2.5 la richiesta totale di τ_2 è $W_2 = 2.5 \leq D_2 = 5$
- τ_2 rispetta la deadline





Analisi del tempo di risposta - Esempio

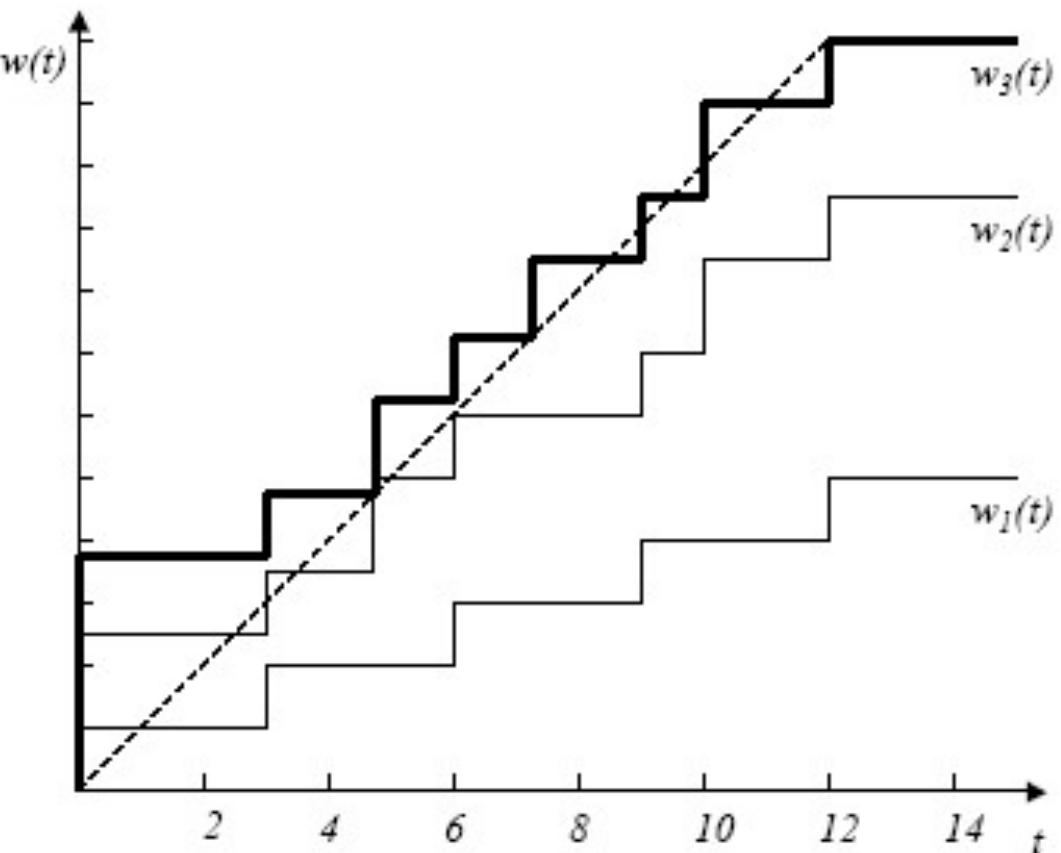
$$\tau_1 = (3, 1)$$

$$\tau_2 = (5, 1.5)$$

$$\tau_3 = (7, 1.25)$$

$$\tau_4 = (8, 0.5)$$

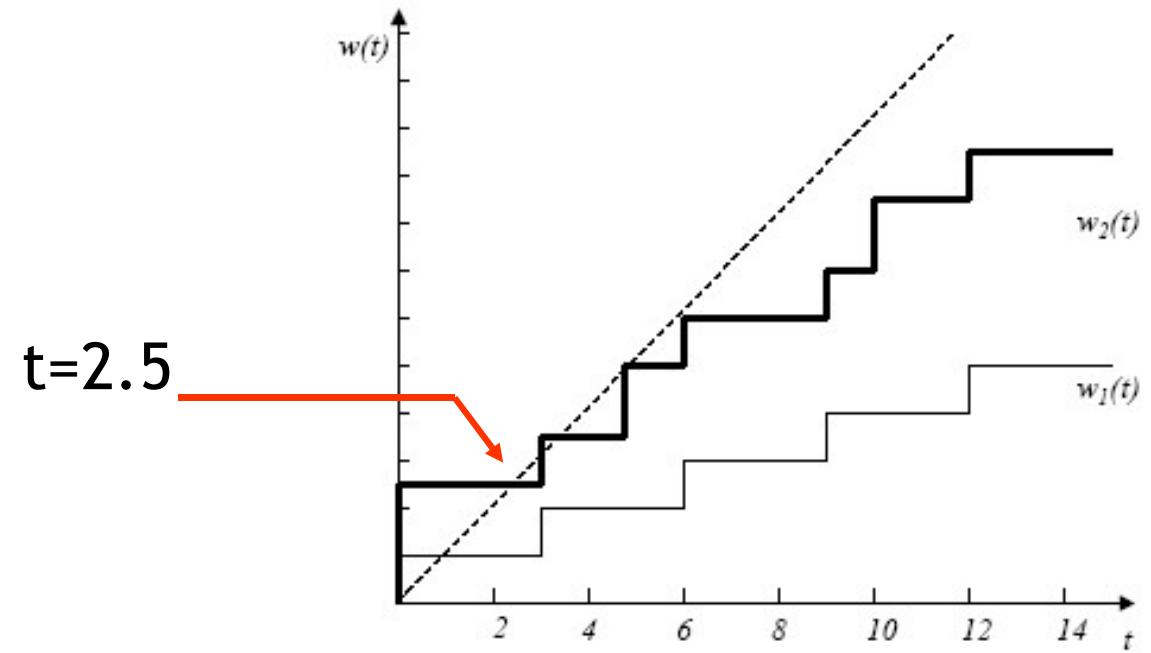
- All'istante 4.75 la richiesta totale di τ_3 è $W_3 = 4.75 \leq D_3 = 7$
- τ_3 rispetta la deadline





Analisi del tempo di risposta - Esempio

- Il punto di intersezione tra $W_i(t)$ e la retta $y=t$ individua l'istante in cui τ_i completa a partire dal suo istante critico

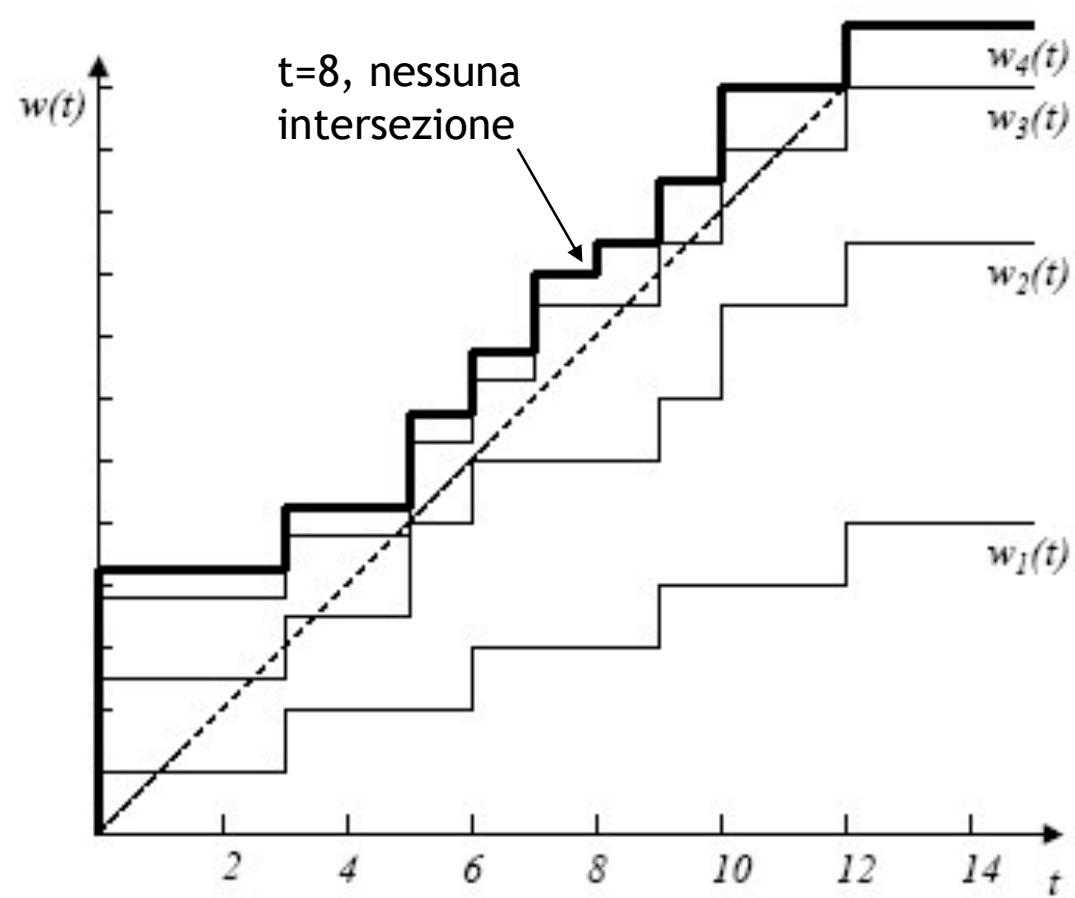




Analisi del tempo di risposta - Esempio

- nessuna intersezione tra W_4 e $y=t$ entro D_4
- τ_4 non rispetta la deadline

$$\begin{aligned}\tau_1 &= (3, 1) \\ \tau_2 &= (5, 1.5) \\ \tau_3 &= (7, 1.25) \\ \tau_4 &= (8, 0.5)\end{aligned}$$





Analisi del tempo di risposta - Esempio

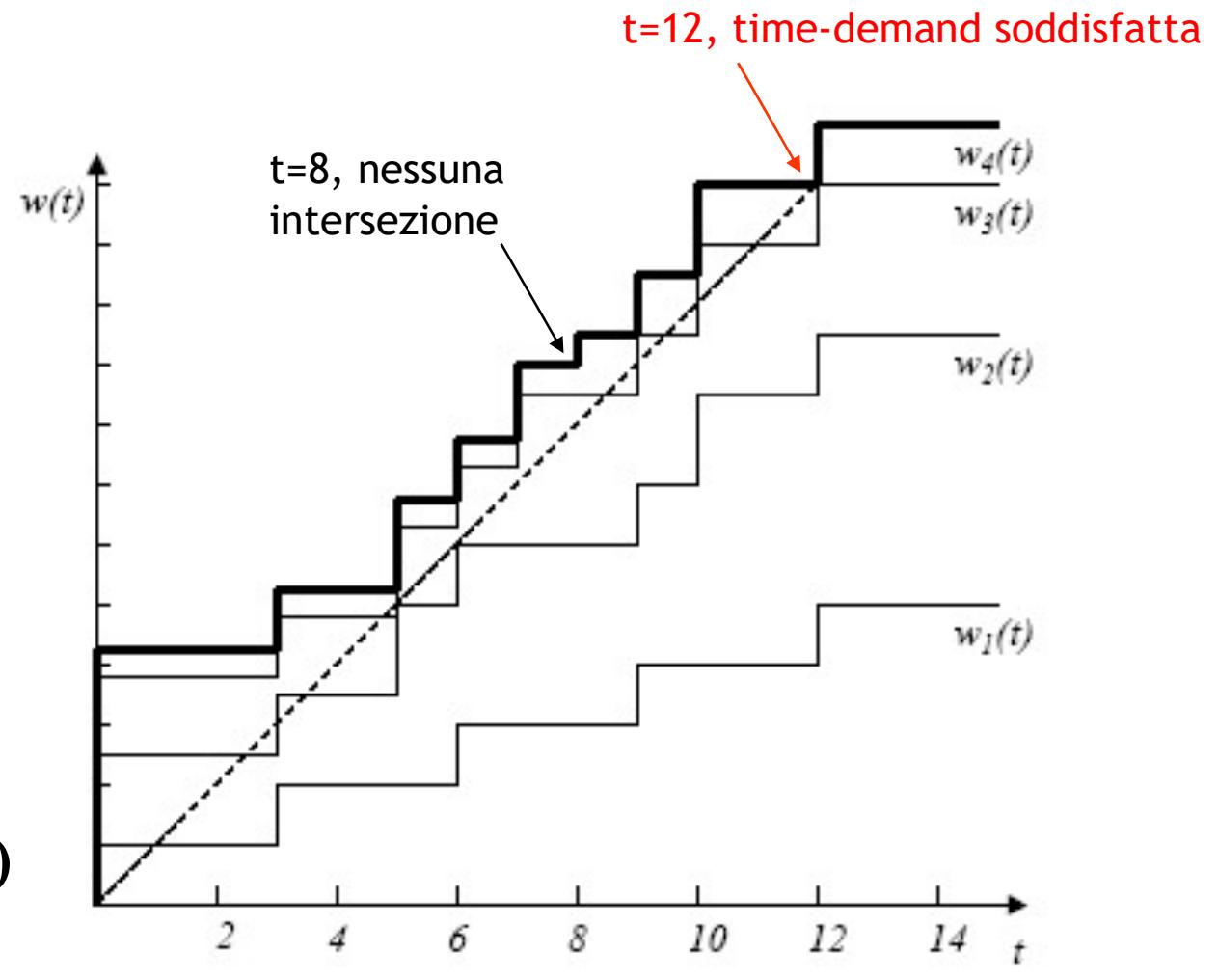
- nessuna intersezione tra W_4 e $y=t$ entro D_4
- τ_4 non rispetta la deadline
- $W_4(t)$ tocca $y=t$ in $t=12$, quindi entro 1.5 periodi recupera e si mette in pari
- backlog max di $\tau_4 = 1$

$$\tau_1 = (3, 1)$$

$$\tau_2 = (5, 1.5)$$

$$\tau_3 = (7, 1.25)$$

$$\tau_4 = (8, 0.5)$$





Analisi del tempo di risposta

Task set modificato:

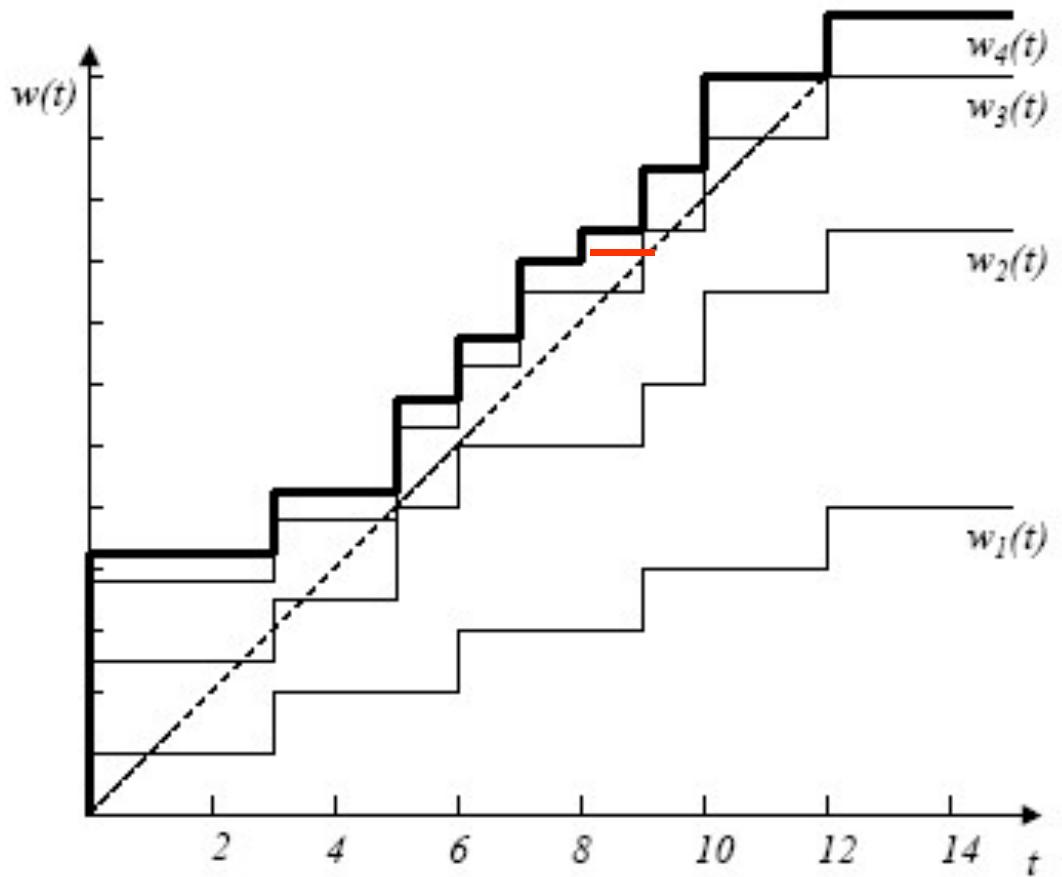
$$\tau_1 = (3, 1)$$

$$\tau_2 = (5, 1.5)$$

$$\tau_3 = (7, 1.25)$$

$$\tau_4 = (9, 0.5)$$

all'istante $t=9$ la richiesta totale di τ_4 è $W_4=9$ e la deadline è soddisfatta





Analisi del tempo di risposta

- Esercizio: Modificare i parametri di τ_4 nell'esempio precedente
 - $\tau_4=(10,1)$ -- ce la fa?
 - $\tau_4=(12,1)$ -- ce la fa?



Analisi del tempo di risposta

- Osservazioni:
- La funzione di time-demand di ogni task è una *scalinata*
- La funzione sale negli *istanti multipli interi* dei periodi dei task a più alta priorità e del task stesso
- Immediatamente prima della salita di un gradino, la differenza tra tempo di processore richiesto e tempo disponibile, $W_i(t) - t$, è al valore minimo dall'alzata precedente
- Se non interessa calcolare il massimo tempo di risposta ma *solo se il task è schedulabile*, è sufficiente verificare se $W_i(t) \leq t$ in questi istanti



Algoritmo di analisi del tempo di risposta

- Calcoliamo $W_i(t) = C_i + \sum_{k=1, i-1} \lceil t/T_k \rceil C_k$
- Verifichiamo se $W_i(t) \leq t$ per i valori $t=jT_k$, con $k=1, 2, \dots, i$, e $j=1, 2, \dots, \lfloor \min(T_i, D_i)/T_k \rfloor$
k seleziona i task a priorità maggiore di τ_i
j identifica i rilasci che interferiscono nell'intervallo critico
- Se la diseguaglianza è soddisfatta in almeno un istante, τ_i è schedulabile
- Costo: $O(n q_{n,1})$, ove $q_{n,1} = T_n/T_1$



Algoritmo di analisi del tempo di risposta

- Nell'esempio: $\tau_1=(3,1)$, $\tau_2=(5,1.5)$, $\tau_3=(7,1.25)$, $\tau_4=(9,0.5)$
- τ_1 e τ_2 sono garantiti
- Per τ_3 dobbiamo verificare $W_3(t) \leq t$ negli istanti:
 $k=1: \lfloor T_3/T_1 \rfloor = \lfloor 7/3 \rfloor = 2 \quad \rightarrow t=3, t=6$
 $k=2: \lfloor T_3/T_2 \rfloor = \lfloor 7/5 \rfloor = 1 \quad \rightarrow t=5$
 $k=3: \lfloor T_3/T_3 \rfloor = 1 \quad \rightarrow t=7$
- L'insieme degli istanti in cui esaminare $W_3(t)$ è $(3,5,6,7)$
- Si verifica che $W_3(t) \leq t$ non è soddisfatta per $t=3$ ma è soddisfatta per $t=5$
- Il massimo tempo di risposta di τ_3 è compreso tra 3 e 5. Poiché $R_3 \leq 5$, $R_3 \leq D_3$ e τ_3 è garantito



Analisi del tempo di risposta

- Il metodo è *robusto rispetto a variazioni*, anche transitorie, dei parametri dei task:
- Se il *tempo di esecuzione* di qualche job di τ_i è *inferiore al WCET* considerato per l'analisi, la curva $W_j(t)$ di tutti i job a priorità inferiore a τ_i si abbassa, e quindi aumenta la possibilità di soddisfare il vincolo
- Se l'*intervallo tra due rilasci* di qualche job di τ_i è *superiore al periodo* indicato, la curva $W_j(t)$ dei job a priorità inferiore sale più lentamente



Calcolo del tempo di risposta

- L'analisi del tempo di risposta consente di aggiungere, sulle curve di time-demand, intervalli di non revocabilità dei job e situazioni di autosospensione
- → consente analisi di schedulabilità sotto ipotesi più generali
- Nel caso più semplice, è possibile calcolare il tempo di risposta tramite un algoritmo iterativo (Audsley)



Calcolo del tempo di risposta

- Algoritmo iterativo:
- Si inizia con una stima pari al tempo di esecuzione C_i

$$\begin{cases} R_i^0 = C_i \\ R_i^s = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

- Si itera fino a quando non si ha $R_i^s = R_i^{s-1}$ oppure $R_i^s > D_i$
 - Se $R_i^s = R_i^{s-1}$ e $R_i^s \leq D_i$ il task i è garantito
 - Se $R_i^s > D_i$ il task non è garantito



Calcolo del tempo di risposta - Esempio

- $\tau_1=(3,1)$, $\tau_2=(5,1.5)$, $\tau_3=(7,1.25)$, $\tau_4=(9,0.5)$
- Per τ_3 , sostituendo in $W_i(t) = C_i + \sum_{k=1,i-1} \lceil t/T_k \rceil C_k$ si ha:
$$R_3^* = 1.25 + 1 \lceil R_3/3 \rceil + 1.5 \lceil R_3/5 \rceil$$
- $R_3^0 = 1.25$
- $R_3^1 = 1.25 + 1 \lceil 1.25/3 \rceil + 1.5 \lceil 1.25/5 \rceil = 1.25 + 1 + 1.5 = 3.75$
- $R_3^2 = 1.25 + 1 \lceil 3.75/3 \rceil + 1.5 \lceil 3.75/5 \rceil = 1.25 + 2 + 1.5 = 4.75$
- $R_3^3 = 1.25 + 1 \lceil 4.75/3 \rceil + 1.5 \lceil 4.75/5 \rceil = 1.25 + 2 + 1.5 = 4.75$
- $\rightarrow R_3 = 4.75 \leq D_3$, τ_3 garantito



Analisi del tempo di risposta

- Esercizi: applicare i diversi metodi nei seguenti casi
 - $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.5)$, $\tau_3 = (7, 1.25)$, $\tau_4 = (8, 0.5)$
 - $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.5)$, $\tau_3 = (7, 1.25)$, $\tau_4 = (9, 0.5)$
 - $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.5)$, $\tau_3 = (7, 1.25)$, $\tau_4 = (10, 1)$
 - $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.5)$, $\tau_3 = (7, 1.25)$, $\tau_4 = (12, 1)$



Sezioni di codice non revocabili

- Alcuni job possono essere non revocabili o avere porzioni non revocabili
- Definizione: *porzione non revocabile*
 ρ_i = la più lunga sezione non revocabile dei job di τ_i
- Definizione: Un *job* si dice *bloccato* se subisce una *inversione di priorità*, ovvero se è ritardato nell'esecuzione da un job a *priorità inferiore*
- Per verificare la schedulabilità di un task τ_i occorre considerare:
 - Tutti i task a più alta priorità, e
 - Le porzioni non revocabili dei task a più bassa priorità

Analisi di schedulabilità con porzioni non revocabili



- Definizione: il *tempo di blocco* B_i del task τ_i è il più lungo intervallo di tempo per cui un job di τ_i può essere bloccato da job a priorità inferiore.
- Per algoritmi a priorità statica, ordinando i task per priorità decrescente vale:

$$B_i = \max_{i+1 \leq k \leq n} \rho_k$$

- Funzione di time-demand in presenza di blocco:
$$W_i(t) = C_i + B_i + \sum_{k=1, i-1} \lceil t/T_k \rceil C_k \quad \text{per } 0 < t \leq \min(D_i, T_i)$$
- (Per algoritmi a priorità dinamica -> teorema di Baker, dopo)



Analisi del tempo di risposta con porzioni non revocabili

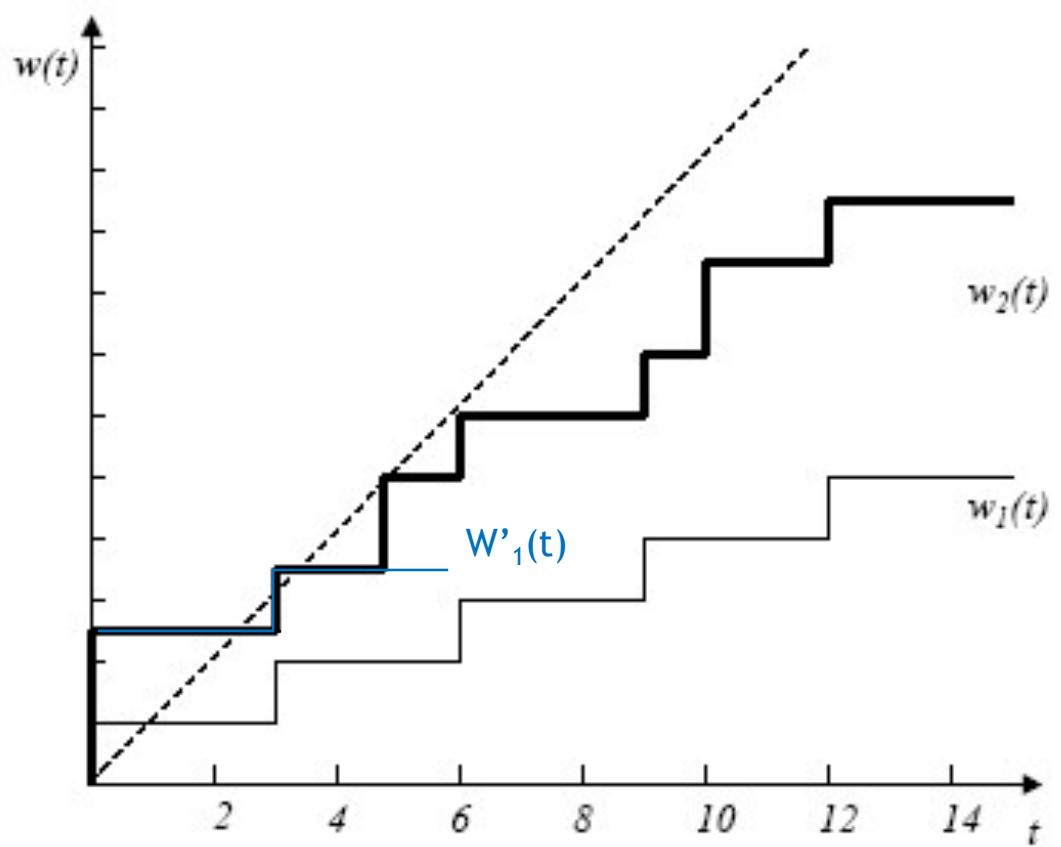
$$\tau_1 = (3, 1)$$

$$\tau_2 = (5, 1.5) \text{ non revoc.}$$

$$\tau_3 = (7, 1.25)$$

$$\tau_4 = (9, 0.5)$$

- La time-demand totale di τ_2 non cambia
- La time-demand totale di τ_1 $w'_1(t)$ aumenta di $B_1=1.5$



Analisi del tempo di risposta con porzioni non revocabili



- In questo caso particolare la time demand cambia solo per τ_1 . Le time demand di τ_2 , τ_3 , τ_4 invece non cambiano
- Si osserva che τ_1 è comunque garantito, poiché completa in $t=2.5$ nel caso peggiore, così come accade a τ_2 . Per i rimanenti task non cambia nulla: semplicemente i primi due task possono scambiarsi di priorità
- Se invece fosse τ_3 a non essere revocabile ...



Bound di utilizzazione in presenza di porzioni non revocabili

- Per sistemi a priorità fissa
- Occorre considerare il tempo di blocco che può subire ogni task
- Per il task τ_i :

$$C_1/T_1 + C_2/T_2 + \dots + (C_i + B_i)/T_i = \sum_{k=1,i} C_k/T_k + B_i/T_i \leq U_x(i)$$

- Ove ad esempio $x=LL$ o $x=KM$ per RM
- Ogni task può subire blocchi di durata diversa e da task diversi, pertanto il test deve essere effettuato *un task alla volta* con priorità decrescente

Scheduling deadline-driven con porzioni non revocabili



- Per sistemi a priorità dinamica (EDF)
- Teorema: In un insieme di task schedulati con EDF, un job J_k con deadline relativa D_k può bloccare un job J_i con deadline relativa D_i solo se $D_k > D_i$ [Baker, 1991]
- Indicizzando i task in base alle deadline relative, il tempo di blocco di ciascun task periodico dovuto a porzioni non revocabili è ancora: $B_i = \max_{i+1 \leq k \leq n} \rho_k$
ove i task sono ordinati per deadline relative crescenti

Scheduling deadline-driven con porzioni non revocabili



- Dimostrazione [Baker, 1991]:
- J_k può bloccare J_i solo se ha priorità inferiore. Poiché le priorità sono assegnate in modo EDF questo significa che J_k ha una deadline posteriore, ovvero $d_k > d_i$
- Inoltre, quando il job a priorità maggiore J_i è stato rilasciato J_k doveva già essere in esecuzione per bloccare J_i , Questo significa che J_k è stato rilasciato prima, ovvero $r_k < r_i$
- Le due diseguaglianze sono soddisfatte solo se $D_k > D_i$ □



Scheduling deadline-driven con porzioni non revocabili

- Un task τ_i con un tempo di blocco B_i è schedulabile con altri task periodici indipendenti dall'algoritmo EDF su un processore se
$$\sum_{k=1,n} C_k / \min(D_k, T_k) + B_i / \min(D_i, T_i) \leq 1$$
- Il sistema è schedulabile se la condizione è verificata per ogni $i=1, 2, \dots, n$



Scheduling deadline-driven con porzioni non revocabili

- Un task τ_i con un tempo di blocco B_i è schedulabile con altri task periodici indipendenti dall'algoritmo EDF su un processore se
$$\sum_{k=1,n} C_k / \min(D_k, T_k) + B_i / \min(D_i, T_i) \leq 1$$
- Il sistema è schedulabile se la condizione è verificata per ogni $i=1, 2, \dots, n$

risultato negativo per EDF: la condizione considera tutti i task !



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Scheduling di task periodici mediante algoritmi a priorità dinamica

Scheduling priority-driven con algoritmi a priorità dinamica



- EDF è il principale algoritmo di schedulazione dinamica priority-driven
- Algoritmi alternativi: FIFO, LST, LRT
 - presentano vari limiti e inconvenienti ...
- → Ci focalizziamo su EDF



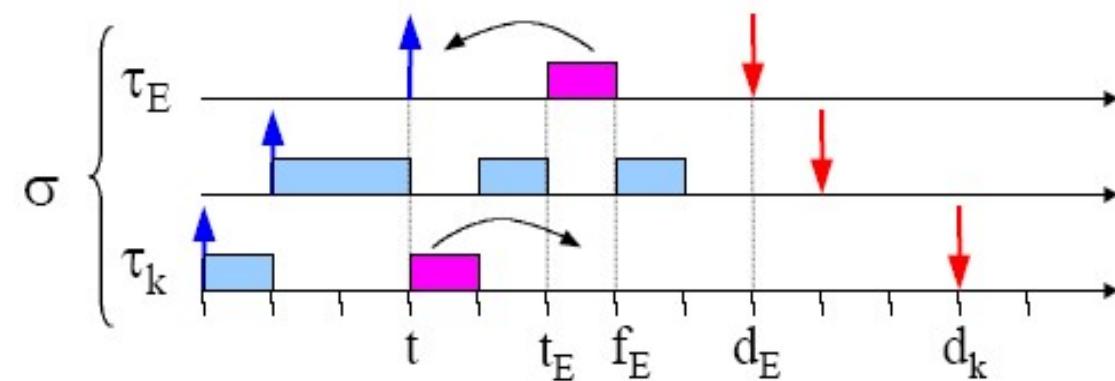
Algoritmo EDF

- EDF assegna la priorità *ai singoli job* dei task periodici in funzione delle loro *deadline absolute* d_i
- EDF è il più importante tra gli algoritmi di scheduling dinamici: offre elevate prestazioni e realizzabilità pratica
- EDF è ottimo: se esiste una schedule fattibile per un insieme di task Γ , anche EDF produce una schedule fattibile per Γ (Dertouzos, 1974)
- La dimostrazione dell'ottimalità di EDF avviene per trasformazione della schedule in senso EDF



Ottimalità di EDF

- Schedule σ non EDF: all'istante t non viene eseguito il job di τ_E la cui deadline d_E è più prossima, ma un job di τ_k con deadline $d_k > d_E$; τ_E viene invece eseguito in $t_E > t$
- Trasformiamo σ in una nuova schedule σ' :
 $\sigma'(t) = \sigma(t_E)$, $\sigma'(t_E) = \sigma(t)$
- La fattibilità è preservata: $f_k = f_E \leq d_E \leq d_k$





Utilizzazione schedulabile per EDF

- Utilizzazione schedulabile di EDF: $U_{lub}(EDF)=1$
- Teorema: Un insieme di n task periodici è schedulabile dall'algoritmo EDF se e solo se l'utilizzazione richiesta è ≤ 1 (Liu e Layland, 1973):

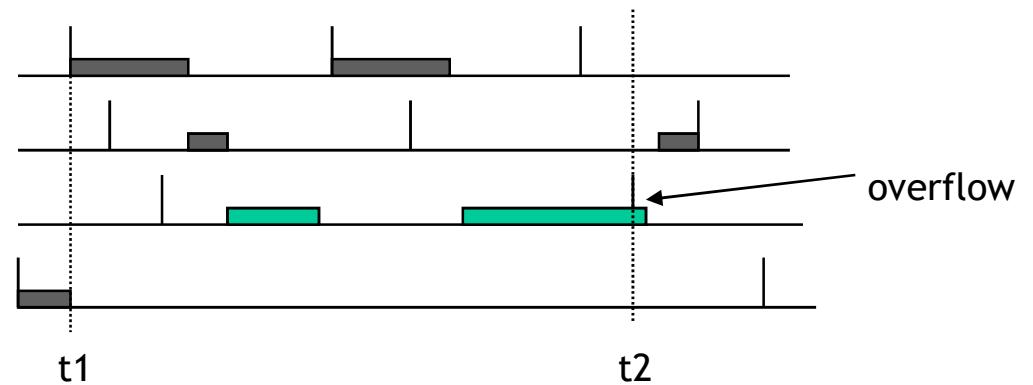
$$\sum_{i=1,n} C_i / T_i \leq 1$$

- Solo se: ovvio.
- Se: ... %



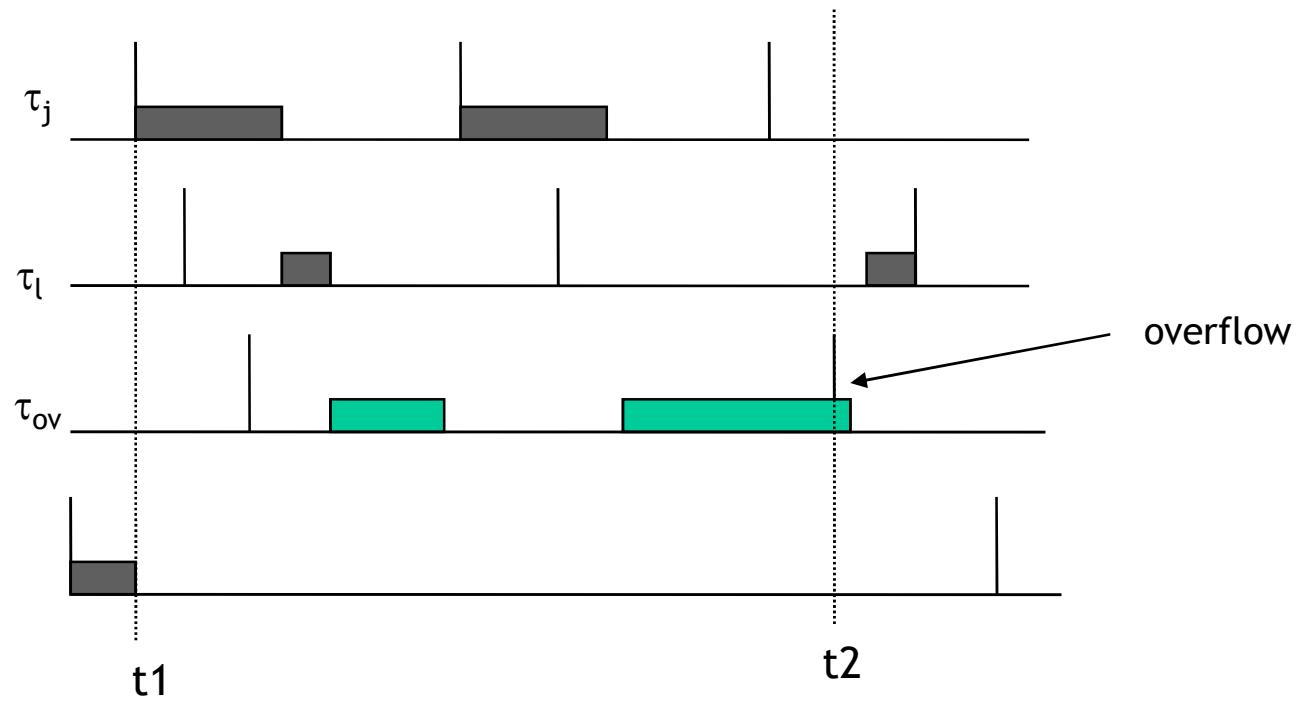
Utilizzazione schedulabile per EDF

- Supponiamo che $U \leq 1$ e che l'insieme di n task periodici non sia schedulabile da EDF: si verifica un overflow in t_2
- Sia $[t_1, t_2]$ il più lungo intervallo di utilizzazione continua prima dell'overflow tale che solo job con deadline $d_j \leq t_2$ sono eseguiti in $[t_1, t_2]$
- t_1 è l'istante di rilascio di un job:





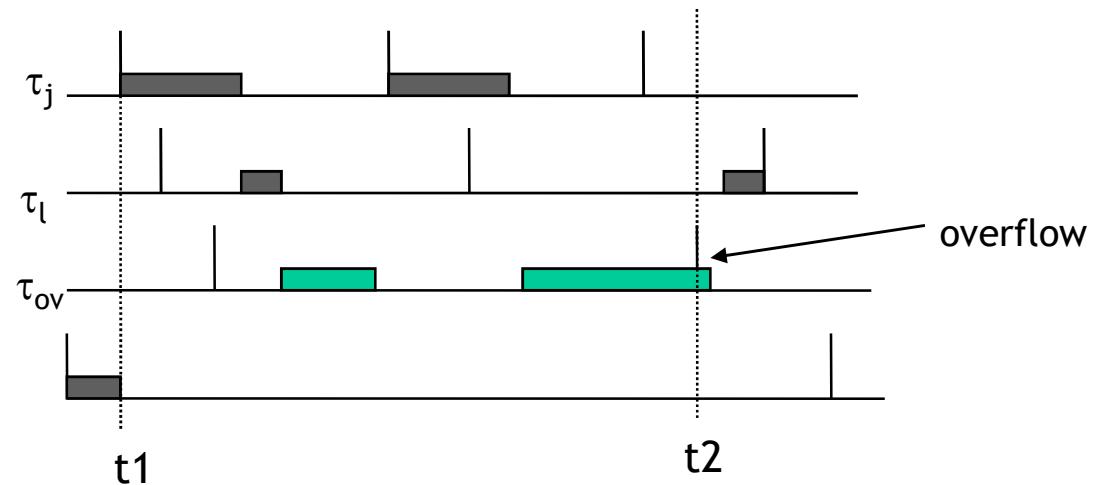
Utilizzazione schedulabile per EDF





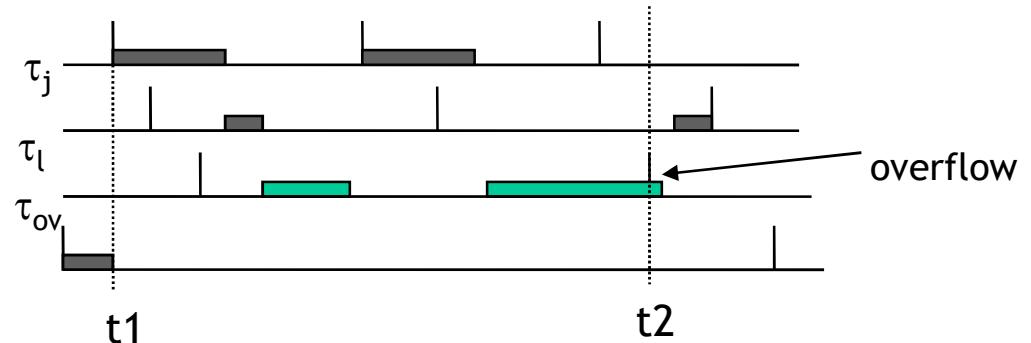
Utilizzazione schedulabile per EDF

- $C_p(t_1, t_2)$ tempo di esecuzione totale richiesto dai task periodici in $[t_1, t_2]$; K =insieme dei *job* rilasciati a partire da t_1 e la cui deadline cade entro t_2
- $C_p(t_1, t_2) = \sum_{k \in K} C_k = \sum_{i=1, n} \lfloor (t_2 - t_1) / T_i \rfloor C_i$





Utilizzazione schedulabile per EDF



- Vale:

$$\begin{aligned} Cp(t_1, t_2) &= \sum_{i=1,n} \lfloor (t_2 - t_1) / T_i \rfloor C_i \leq \sum_{i=1,n} ((t_2 - t_1) / T_i) C_i = \\ &= (t_2 - t_1) U \end{aligned}$$

- Poichè in t_2 viene mancata una deadline:

$$(t_2 - t_1) < Cp(t_1, t_2) \leq (t_2 - t_1)U$$

- Ovvero, $U > 1$, il che è una contraddizione



Realizzazione di EDF

- Con RM o DM la schedulazione è statica e a tempo di esecuzione non c'è overhead per stabilire la priorità dei job
- Con EDF e altri algoritmi dinamici occorre stabilire a tempo di esecuzione la priorità relativa dei job
- Con EDF il costo è quello di un ordinamento di un gruppo di job ($O(n \log n)$), ovvero di inserimento di un job in una lista ordinata ($O(n)$)

- EDF presenta una complessità realizzativa ragionevole → algoritmo di scheduling largamente usato!



EDF con deadline relative distinte dai periodi

- *Densità* di un task $\tau_i = (T_i, C_i, D_i)$: $\Delta_i = C_i / \min(D_i, T_i)$
- Condizione *sufficiente* affinchè un insieme di N task periodici, anche con deadline relative diverse dai periodi, sia schedulabile con l'algoritmo EDF è che:

$$\Delta = \sum_{i=1, N} C_i / \min(D_i, T_i) \leq 1$$

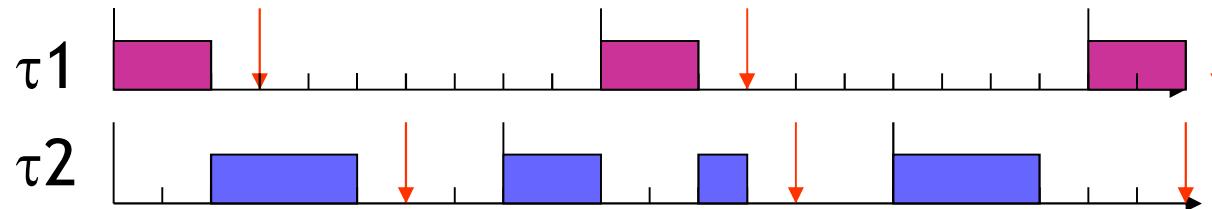
- Δ è la *densità totale* dell'insieme di task
- E' una condizione sufficiente ma non necessaria



EDF con deadline relative distinte dai periodi

- Esempio (già visto con algoritmo DM):

$$\tau_1 = (10, 2, 3) \quad \tau_2 = (8, 3, 6) \quad [\tau_i = (T_i, C_i, D_i)]$$



- $\Delta = \sum_{i=1,N} C_i / \min(D_i, T_i) = 2/3 + 3/6 = 1.16 > 1$
- --> I task non sono garantiti dal bound di densità di EDF (condizione solo sufficiente)
- NB: La schedule in figura è anche una schedule EDF



EDF con deadline relative distinte dai periodi

- Q: I task dell'esempio precedente sono garantiti in caso di schedulazione EDF?

Esercizi: analisi di schedulabilità con uso dei bound



- (A) Task set: $\Gamma = \{ \tau_1 = (10, 6), \tau_2 = (25, 5), \tau_3 = (50, 5) \}$
- (B) Task set: $\Gamma = \{ \tau_a = (10, 2), \tau_b = (8, 2), \tau_c = (25, 5), \tau_d = (5, 1) \}$

- Quesiti:
- task set schedulabile? (precisare quale algoritmo e quale bound di utilizzazione)
- singoli task individualmente garantiti? (in base ad algoritmo e bound)
- schedulabilità e garanzia in senso assoluto: combiniamo i diversi strumenti di analisi



Task armonici

- Insieme di *task armonici*: data una coppia di task qualsiasi dell'insieme, il periodo di un task è multiplo del periodo dell'altro
- Sono detti anche insiemi di task *semplicemente periodici*
- Teorema (Lehoczky et al. 1989):
Un insieme di task armonici indipendenti, con preemption ovunque consentita e le cui deadline coincidono con il periodo, è schedulabile dall'algoritmo RM se e solo se la loro utilizzazione totale non supera 1
- Ovvero: Per insiemi di task armonici, $U_{lub}(RM)=1$



U_{lub} per task armonici

- Teorema: Per un insieme di task armonici $U_{lub}(RM)=1$.
- Dim. per contraddizione: Per un insieme di task armonici con $U \leq 1$, schedulato in modo RM, un task manca la deadline
 - Ipotizziamo che τ_i manchi la propria deadline all'istante t ; t è un multiplo intero di T_i e di tutti i T_k con $k \leq i-1$
 - Tempo totale per completare i job con deadline entro t :
$$\sum_{k=1,i} (t/T_k) C_k = t \sum_{k=1,i} C_k / T_k = t U_i$$
 - ove U_i è il fattore di utilizzazione richiesto dai primi i task a più alta priorità e (t/T_k) è il numero di attivazioni di ciascun task τ_k entro il periodo t
 - Se il job manca la propria deadline \rightarrow il tempo totale richiesto era insufficiente: $t U_i > t \rightarrow U_i > 1 \rightarrow U > 1$ (c.v.d.)



Insiemi di task armonici - Esempio

$$\tau_1: T_1 = 10 \quad C_1 = 3 \quad U_1 = 0.300$$

$$\tau_2: T_2 = 30 \quad C_2 = 2 \quad U_2 = 0.067 \quad U_1+U_2 = 0.367$$

$$\tau_3: T_3 = 30 \quad C_3 = 5 \quad U_3 = 0.167 \quad U_1+U_2+U_3 = 0.534$$

$$\tau_4: T_4 = 300 \quad C_4 = 100 \quad U_4 = 0.334 \quad U_1+U_2+U_3+U_4 = 0.868$$

- L'insieme di task (τ_1, τ_2, τ_3) è certamente schedulabile, perché $U_1+U_2+U_3 < U^*(3) = 0.779$
- Viceversa: $U_1+U_2+U_3+U_4 > U^*(4) = 0.756$
- Tuttavia non è necessario operare alcuna analisi dell'intervallo critico di τ_4 perché i task sono armonici



Insiemi di task armonici

- Talvolta esiste un margine di flessibilità nello specificare i periodi dei task, specie di quelli a minore priorità
- Conviene, se possibile, rendere i task armonici, per conseguire $U = 1$
- Può essere utile anche rendere armonici solo *alcuni* task, quelli per i quali esiste margine di flessibilità (per semplificare analisi e dispatching, ...)

Test di schedulabilità dipendenti dai parametri dei task



- Il test basato sull'utilizzazione schedulabile è particolarmente efficiente e può essere integrato in un modulo di accettazione
- L'elaborazione è garantita se (solo sufficiente):

$$U_p \leq n(2^{1/n} - 1)$$

- Il test di Liu e Layland per RM, basato esclusivamente sul numero di task periodici, è solo sufficiente e risulta spesso troppo prudente
- Può essere migliorato, in alcune situazioni, considerando anche i parametri dei task schedulati



Test di Kuo e Mok ('91)

- Un insieme S di N task periodici con $D_i = T_i$ è schedulabile da RM se può essere *partizionato* in N_h sottoinsiemi disgiunti S_i tali che all'interno di ogni partizione i task risultano *semplicemente periodici*, ed è schedulabile da RM l'insieme di task S' in cui ad ogni partizione corrisponde un task di periodo minimo tra i task della partizione e fattore di utilizzazione pari alla somma delle utilizzazioni dei task della partizione.
- Uh?
- Ovvero: $U_{lub}(RM) = N_h(2^{1/N_h} - 1)$



Test di Kuo e Mok - esempio

- Set iniziale: $\tau_1=(10,5)$, $\tau_2=(25,5)$, $\tau_3=(50,5)$
- $U_1=0.5$, $U_2=0.2$, $U_3=0.1$, $\sum U_j = 0.8 > U_{LL}(3)=0.78$
- Partizione: $(\tau_1), (\tau_2, \tau_3)$
- Set trasformato:
 $\tau'_1=(T=10, U=0.5)$, $\tau'_2=(T=25, U=0.3)$
- Per il set trasformato vale $N_h=2$: $\sum U_j = 0.8 < U_{LL}(N_h=2)=0.828$
- → il set iniziale è garantito dal bound KM se schedulato in modo RM
- In generale è sufficiente verificare che $U \leq N_h(2^{1/N_h} - 1)$



Applicazione del test di Kuo e Mok

- In generale, per applicare il bound di Kuo-Mok *non è necessario costruire alcun set equivalente!*
- E' sufficiente calcolare l'utilizzazione richiesta e confrontare tale valore con il bound modificato tenendo conto del teorema di Kuo e Mok
- L'utilizzazione richiesta non cambia applicando il bound di Liu e Layland o quello di Kuo e Mok (né altri bound)!
- In altre parole: occorre sempre considerare i task con priorità rate-monotonic decrescente applicando il bound di Kuo e Mok *al sottoinsieme di task in esame: lo scheduling resta rate-monotonic*



Test di Kuo e Mok - variazione esempio

- Set iniziale: $\tau_1=(10,6)$, $\tau_2=(25,5)$, $\tau_3=(50,5)$
- $U_1=0.6$, $U_2=0.2$, $U_3=0.1$, $\sum U_j = 0.9 > U_{LL}(3)=0.78$
- Partizione: $P_a=\{(\tau_1, \tau_3), (\tau_2)\}$ oppure $P_b=\{(\tau_1), (\tau_2, \tau_3)\}$
 - set trasformato P_a : $\tau'_1=(T=10, U=0.7)$, $\tau'_2=(T=25, U=0.2)$
 - set trasformato P_b : $\tau'_1=(T=10, U=0.6)$, $\tau'_2=(T=25, U=0.3)$
- Per il set trasformato, $N_h=2$: $\sum U_j = 0.9 > U_{LL}(2)=0.828$
→ il set iniziale non è garantito dal bound KM
- Quali task sono individualmente garantiti?
- Ricordarsi: *lo scheduling resta RM!*



Il bound iperbolico

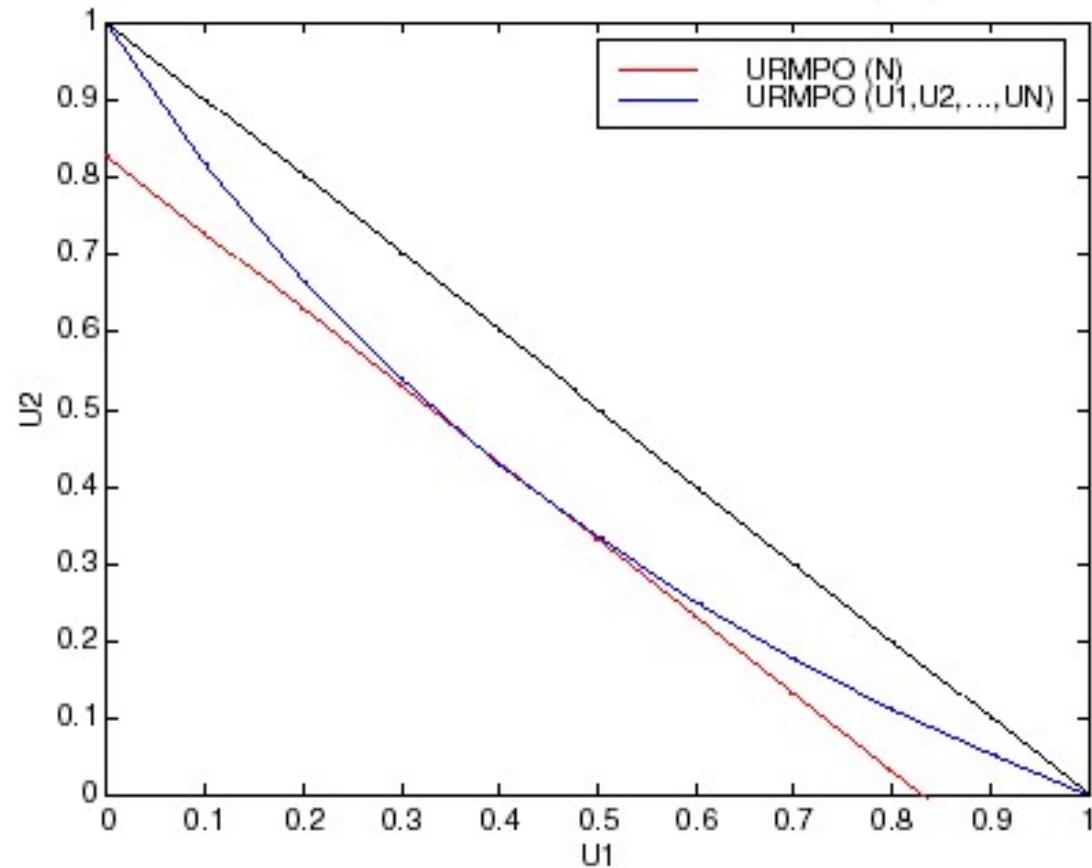
- Hyperbolic Bound (HB) è un bound meno restrittivo sul fattore di utilizzazione disponibile, applicabile per l'algoritmo RM
- Come il bound di Kuo e Mok, richiede di esplicitare *i parametri temporali* dei task periodici
- Meno conservativo del bound di Liu e Layland (che si basa solo *sul numero di task*)
- Un insieme di N task eseguito con priorità RM è garantito se vale:

$$\prod_{(j=1,N)} (1 + U_j) \leq 2$$



Il bound iperbolico

- Per 2 task:





Bound iperbolico - Esempio

- Set iniziale: $\tau_1=(10,5)$, $\tau_2=(25,5)$, $\tau_3=(50,5)$
- $U_1=0.5$, $U_2=0.2$, $U_3=0.1$, $\sum U_j = 0.8 > U_{LL}(3)=0.78$
- Vale: $\prod_{(j=1,3)} (1 + U_j) = 1.98 \leq 2$
- Poiché il bound iperbolico è soddisfatto, l'insieme di task è garantito (dal bound iperbolico) se schedulato in modo RM
- Esercizio: $\tau_1=(10,5)$, $\tau_2=(25,5)$, $\tau_3=(55,6)$



Algoritmo Deadline Monotonic

- Algoritmo DM (Leung e Whitehead, 1982): ordina le priorità in funzione delle *deadline relative* D_i
- *Ottimalità di DM*: Se un insieme di task è schedulabile da un algoritmo a priorità fissa, esso può essere schedulato anche dall'algoritmo DM
- DM opera in *ipotesi più generali* di RM: con *deadline relative arbitrarie* DM è ottimo, mentre RM non lo è
 - ovvero: RM resta ottimo solo se le deadline relative sono proporzionali ai periodi



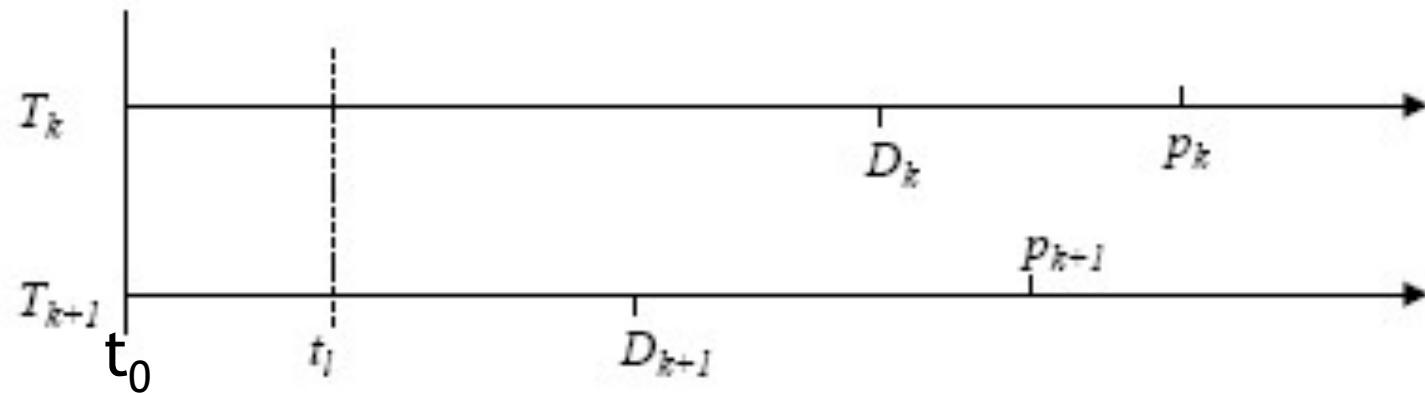
Algoritmo Deadline Monotonic

- Dimostrazione dell'ottimalità di DM:
- Insieme di task $\tau_1, \tau_2, \dots, \tau_n$ a priorità decrescente, schedulabile in modo non DM. Esiste quindi una coppia di task consecutivi τ_k e τ_{k+1} per i quali $D_k > D_{k+1}$
- Consideriamo l'intervallo critico a partire da t_0 di τ_{k+1} : è la situazione peggiore per tutti gli algoritmi a priorità statica. Valutiamo l'orizzonte temporale fino a $D_{k+1} (< D_k)$
- Poiché l'insieme è schedulabile in modo non DM, τ_{k+1} completa entro D_{k+1} ed è preceduto dai job a più alta priorità ed anche dal job (unico nell'intervallo critico) di τ_k



Algoritmo Deadline Monotonic

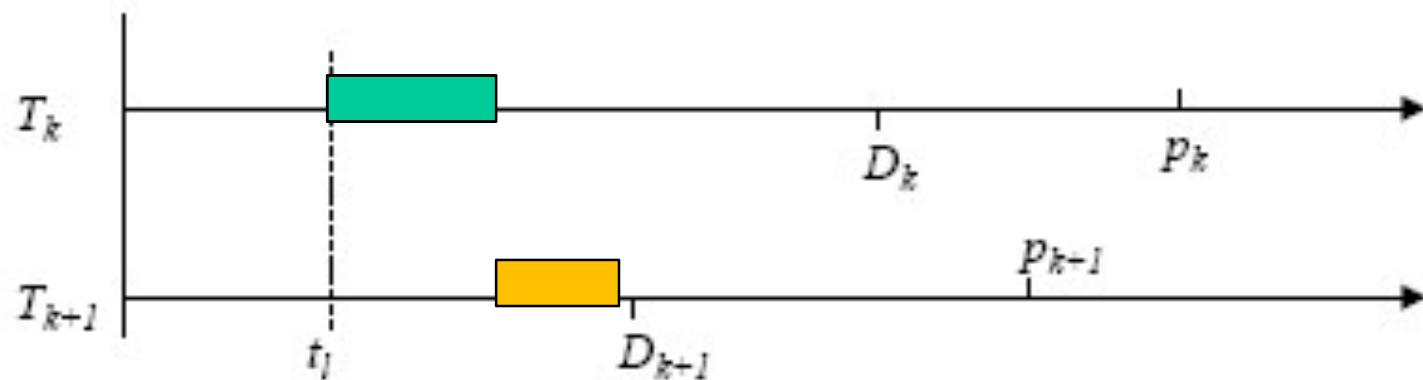
- L'esecuzione di τ_k e τ_{k+1} può essere divisa in più intervalli per le preemption da parte di job a più alta priorità, ma in ogni caso scambiando la priorità di τ_k e τ_{k+1} , e quindi l'ordine di esecuzione negli intervalli che precedono D_{k+1} , l'insieme resta schedulabile





Algoritmo Deadline Monotonic

- Iterando tra tutte le coppie di task non ordinate con priorità DM si completa la dimostrazione
- Nella situazione in figura, scambiando in t_l la priorità tra τ_k e τ_{k+1} l'insieme di task resta schedulabile





Utilizzazione schedulabile per DM

- Per garantire un insieme di n task periodici *con deadline relativa minore o uguale del periodo* $D_i \leq T_i \forall i$, si può utilizzare una variante del test RM:

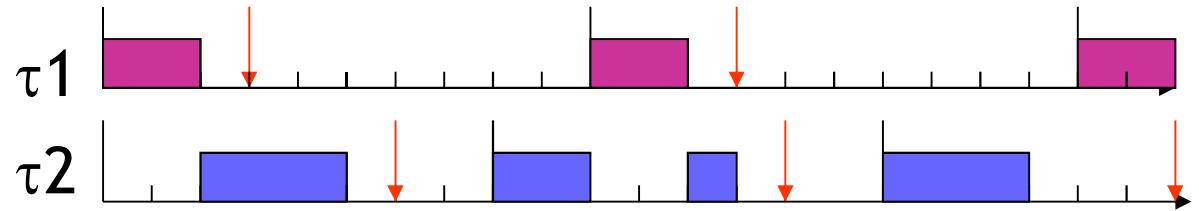
$$\sum_{i=1,n} C_i/D_i \leq n(2^{1/n} - 1) \quad (*)$$

- Questo condizione è solo sufficiente e in generale è *estremamente peggiorativa*
- Se $D_i < T_i$ per qualche i , un insieme di task potrebbe essere schedulabile anche con $\sum_{i=1,n} C_i/D_i > 1$
- Se il test (*) non è soddisfatto, occorre effettuare un'analisi di schedutabilità a partire dall'istante critico nell'asse del tempo



Schedulazione DM

- Esempio: $\tau_1 = (10, 2, 3)$ $\tau_2 = (8, 3, 6)$ $[\tau_i = (T_i, C_i, D_i)]$

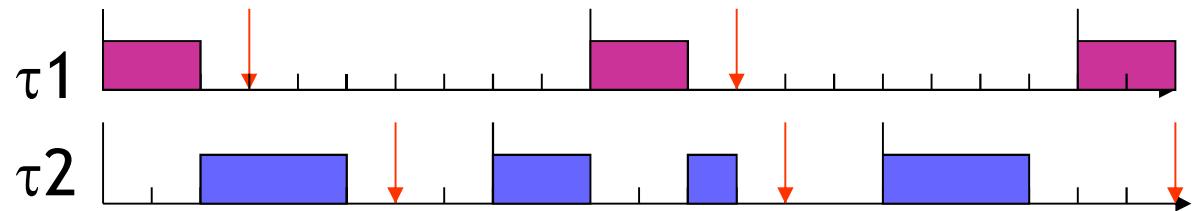


- $\sum_{i=1,n} C_i/D_i = 2/3 + 3/6 = 1.16 > 1$ -- cosa rappresenta?



Non ottimalità di RM nel caso $D_i \neq T_i$

- Dimostrazione:
- Esempio: $\tau_1 = (10, 2, 3)$ $\tau_2 = (8, 3, 6)$ $[\tau_i = (T_i, C_i, D_i)]$



- RM non produce una schedule fattibile (τ_1 manca la deadline) mentre DM sì, pertanto RM non è ottimo in queste ipotesi più generali
- $\sum_{i=1,n} C_i/T_i = 2/10 + 3/8 = 0,575 \leftarrow$ utilizzazione



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Scheduling di task periodici basato su priorità

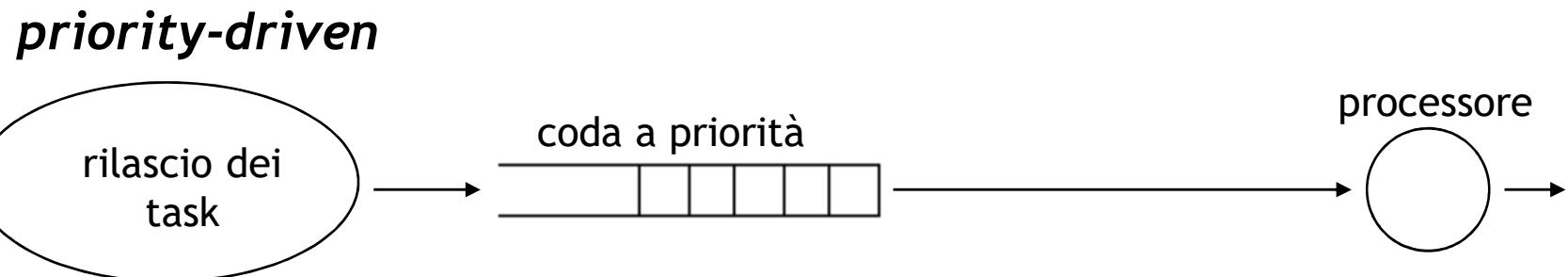
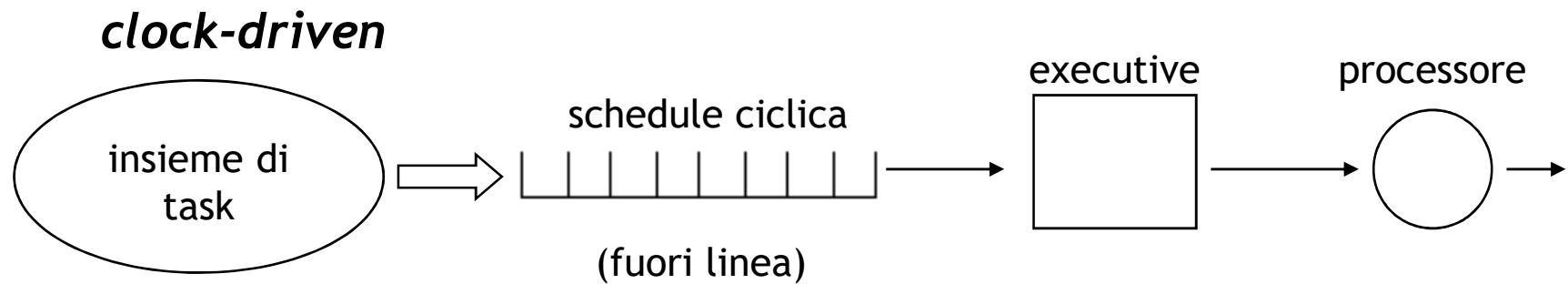
prof. Stefano Caselli

stefano.caselli@unipr.it



Scheduling basato su priorità

- Priority-driven vs. clock-driven:





Scheduling di task periodici basato su priorità

- Ipotesi:
 - task indipendenti
 - task periodici
 - assenza di task aperiodici e sporadici
 - i task sono pronti non appena rilasciati e non si sospendono da soli
 - preemption consentita ovunque
- Le decisioni di scheduling sono prese in corrispondenza al rilascio ed al completamento dei job
 - overhead per cambio di contesto trascurabile
 - numero illimitato di livelli di priorità



Scheduling di task periodici basato su priorità

- Ipotesi di contesto:
- I risultati relativi ad insiemi di task *periodici* valgono anche considerando il periodo come *minimo tempo tra due rilasci* consecutivi di un task
- Il numero di task periodici ammessi nel sistema è fisso o regolato da un modulo di accettazione
- Facciamo riferimento a sistemi uniprocessore, o a sistemi multiprocessore in cui i task sono allocati *staticamente*



Algoritmi a priorità fissa e dinamica

- Uno scheduler priority-driven opera *on-line*: assegna priorità ai job quando vengono rilasciati e li inserisce nella coda dei job pronti in base alla loro priorità
- Algoritmi *a priorità fissa*: tutti i job di un task hanno la stessa priorità → la priorità del task è statica (es.: RM)
- Algoritmi *a priorità dinamica*: possono assegnare priorità diversa ai job di un task → la priorità *del task* è dinamica (es.: EDF)
- Normalmente *i singoli job* hanno comunque una priorità statica



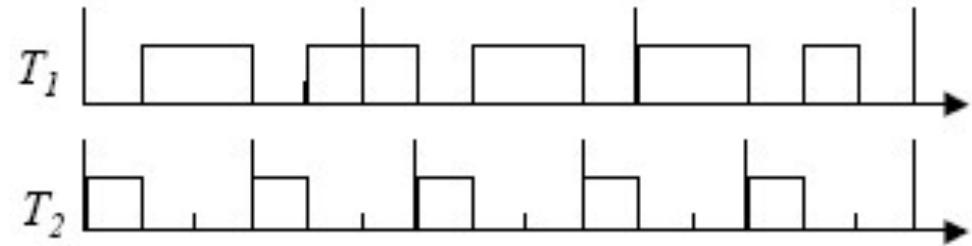
Classificazione degli algoritmi a priorità

- La priorità è attribuita ai job *quando vengono rilasciati* ed inseriti tra i job pronti
- Una volta assegnata, la priorità di un job rispetto non cambia
- Classificazione:
 - algoritmi a *priorità fissa o statica* (per task e job)
 - algoritmi a *priorità dinamica* a livello di task (a priorità fissa a livello di job)
 - algoritmi a priorità dinamica a livello di job (e quindi anche di task) -- poco frequenti



Algoritmo Rate Monotonic

- La priorità è attribuita ai task in base al loro *periodo*: minore è il periodo, maggiore è la priorità
- Esempio: $T_1=(5,3,5)$, $T_2=(3,1,3)$



- T_2 ha sempre priorità su T_1 , e ciò determina alcune situazioni di *preemption*
- *Priorità statica*: la priorità è proporzionale al *rate* del task (l'inverso del periodo)



Algoritmo Rate Monotonic

- Ha senso?
- Se un task periodico ha un *rate elevato* (cioè un periodo più breve), tutti i suoi job avranno scadenze ravvicinate
- → può essere utile attribuire ai job del task una priorità elevata, a fattor comune
- Stessa priorità per tutti i job → *priorità del task*
- Evita manipolazioni della priorità a tempo di esecuzione:
setting priorities? it's cheap but it ain't free

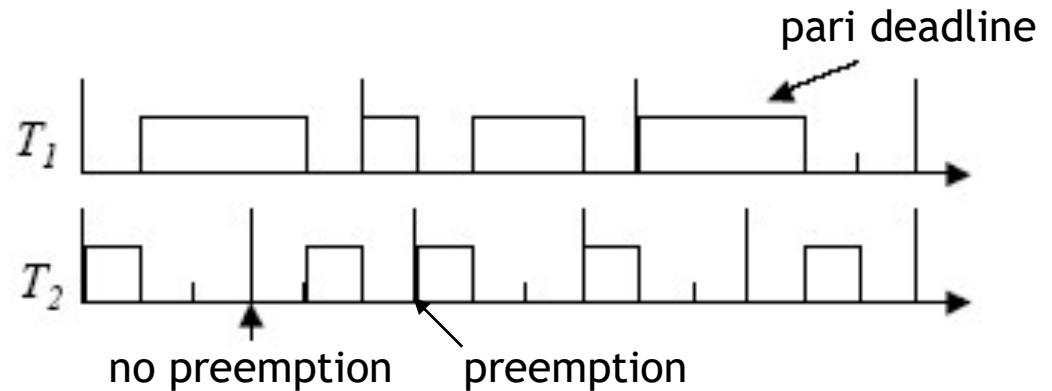


Algoritmo Earliest Deadline First

- La priorità è attribuita ai job in base alla loro *deadline assoluta*: più prossima è la deadline assoluta, maggiore è la priorità

- Esempio:

$$T_1 = (5, 3, 5), T_2 = (3, 1, 3)$$



- La *preemption* si può ancora verificare, in base al diverso criterio di priorità
- *Priorità dinamica*: i job di T_i possono avere priorità relativa diversa rispetto ai job di un altro task periodico T_j

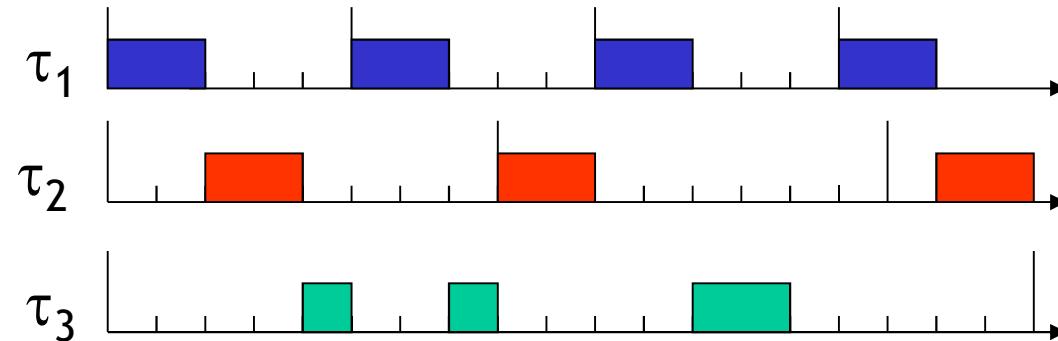


Algoritmo Earliest Deadline First

- La priorità EDF esprime direttamente l'urgenza del job
- Job diversi dello stesso task rilasciati nel tempo hanno priorità diversa
- Se $T_i < D_i$, job diversi dello stesso task possono essere pronti per l'esecuzione, ma la politica EDF ne determina una diversa priorità e i job saranno eseguiti nell'ordine di rilascio
- Ad ogni rilascio di un job lo scheduler ne deve definire la priorità relativa rispetto agli altri job pronti --> per la realizzazione è utile una struttura *callback queue*



Algoritmo Rate Monotonic



- ❑ Negli algoritmi statici, i task a priorità maggiore sono “protetti”
- ❑ I job dei task a priorità non massima possono subire più revocate per periodo in funzione del numero e della frequenza dei task a priorità più alta



Algoritmi statici

-
- La priorità è stabilita in base a caratteristiche intrinseche *del task*, che valgono per tutti i suoi job: periodo, deadline relativa, “importanza”
 - I principali sono *Rate Monotonic* (RM) e *Deadline Monotonic* (DM), che ordina le priorità dei task in base alle deadline relative
 - L’algoritmo RM (Liu e Layland, 1973) è ottimo tra gli algoritmi statici in cui la deadline relativa coincide con il periodo
 - L’algoritmo DM (Leung e Whitehead, 1982) generalizza RM per deadline arbitrarie, ed è a sua volta ottimo in questa ipotesi più generale



Altri algoritmi statici

- Priorità legata all' “importanza” del task? Es.:
 - priorità basata su criticità funzionale
 - ordinamento alfabetico
 - etc.
- Tipicamente non ottimi
- Con alcuni strumenti può ancora essere possibile garantire uno o pochi task
- Non considerano i parametri temporali che determinano l'urgenza del task!
- → Limitiamo l'attenzione ad algoritmi priority-driven in cui la priorità è collegata a parametri temporali



Algoritmi dinamici

- La priorità è stabilita in base a caratteristiche *tempo varianti del task*: ad es. la deadline *assoluta* del job, l'istante di rilascio del job, etc.
- I principali sono *Earliest Deadline First (EDF)* e *Least Slack Time First (LST)*:
- EDF: considera la deadline assoluta, LST: lo slack time residuo
- FIFO e LIFO: considerano l'istante di rilascio del job; danno luogo a risultati scadenti perchè non si basano su un parametro correlato all'*urgenza* del job



Algoritmi dinamici: varianti

-
- *LST non stretto*: valuta gli slack time solo quando un job è rilasciato o completa
 - *LST stretto*: deve valutare continuamente lo slack time del job in esecuzione; quando raggiunge lo slack di job in attesa, la schedulazione prosegue in modo *Round-Robin* per evitare instabilità



Considerazioni realizzative

- Dal punto di vista realizzativo, gli algoritmi statici sono più semplici:
 - la priorità viene attribuita staticamente
 - il job, quando rilasciato, viene direttamente inserito in coda al livello di priorità che gli compete
- Negli algoritmi dinamici occorre rivalutare la priorità relativa ad ogni rilascio del job
 - Ma ... (Spoiler)



Utilizzazione

- Ogni task periodico usa il processore per la frazione di tempo:

$$U_i = \frac{C_i}{T_i}$$

- L'*utilizzazione* totale del processore è:
$$U_p = \sum_{i=1}^n \frac{C_i}{T_i}$$
- Il (*fattore di*) *utilizzazione* U_p è una misura del carico del processore
- Se i task richiedono $U_p > 1$ il processore è *sovraffatto* e l'insieme di task non può essere garantito



Utilizzazione schedulabile

- Definizione:
valore del fattore di utilizzazione totale associato ad un algoritmo tale che ogni insieme di task periodici la cui utilizzazione totale è non superiore a tale valore è schedulabile dall'algoritmo
- ... maggiore l'utilizzazione schedulabile, migliore l'algoritmo
- E' sempre ≤ 1

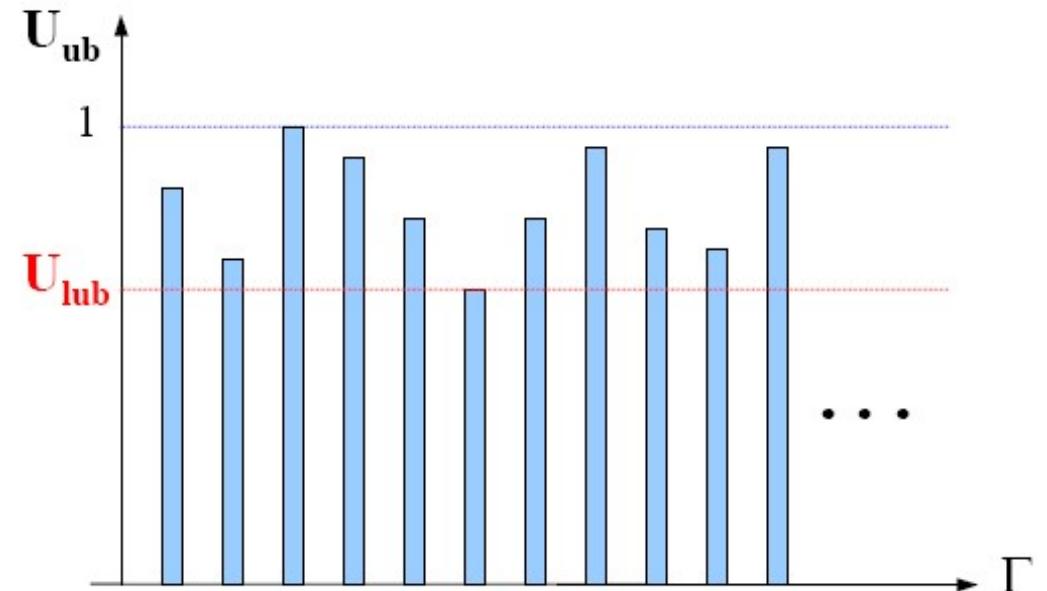
- Denominata anche *Least Upper Bound* del fattore di utilizzazione, U_{lub}



Il Least Upper Bound

- Dato un algoritmo α , il valore massimo di U per cui i task sono schedulabili, U_{ub} , dipende dall'insieme di task Γ
- Il minimo di U_{ub} per cui qualunque insieme di task risulta schedulabile dall'algoritmo α è $U_{lub}(\alpha)$

- Se $U(\Gamma) \leq U_{lub}(\alpha)$, Γ è certamente schedulabile dall'algoritmo α





Utilizzazione schedulabile

- Un test di schedulabilità a basso costo:
- Dato un insieme di task periodici Γ ed un algoritmo di scheduling α , condizione *sufficiente* perchè Γ sia schedulabile da α :

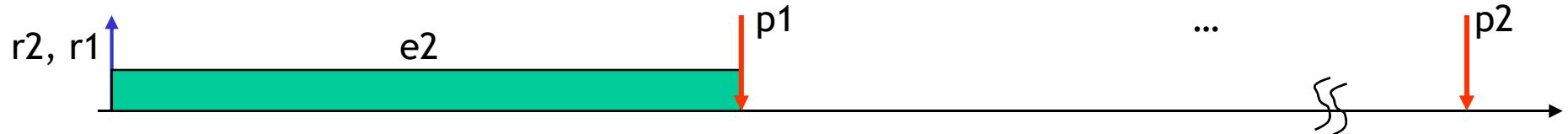
$$U(\Gamma) \leq U_{\text{lub}}(\alpha)$$

- Se il test non è soddisfatto ma $U(\Gamma) \leq 1$, Γ può essere *schedulabile oppure no* da α



Utilizzazione schedulabile per FIFO

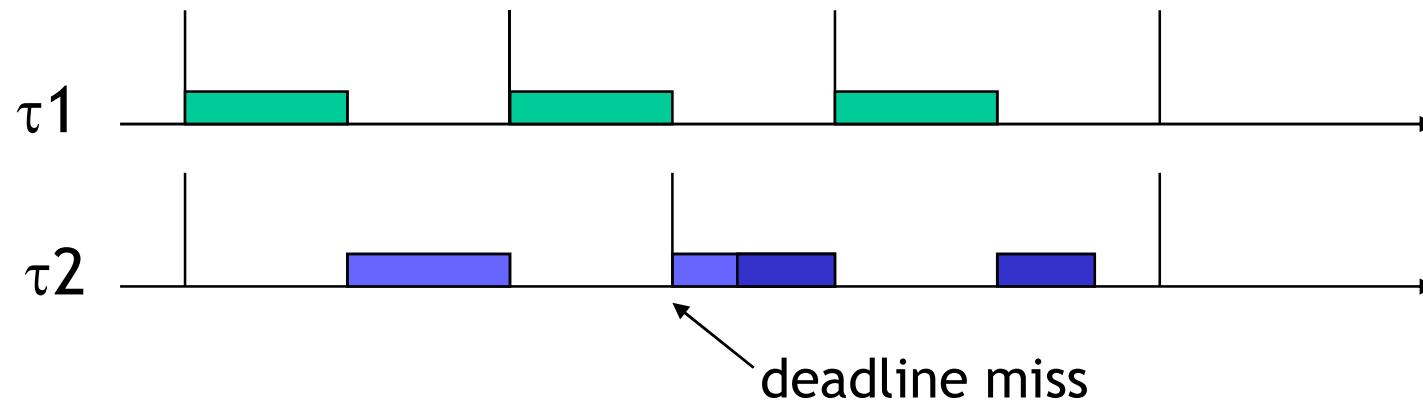
- Teorema: $U_{\text{lub}}(\text{FIFO})=0$
- Dim: Dato un qualunque livello di utilizzazione $\varepsilon > 0$, è possibile costruire un task set con utilizzazione ε ma *non schedulabile* in modo FIFO
- Es.: $\tau_1 = (p_1, e_1) = (p, \varepsilon p/2)$, $\tau_2 = (p_2, e_2) = (2 p/\varepsilon, p)$
 $U_1 = (\varepsilon p/2)/p = \varepsilon/2$, $U_2 = p/(2p/\varepsilon) = \varepsilon/2 \rightarrow U = \varepsilon$
- se τ_1 arriva appena dopo τ_2 , τ_1 manca la deadline





Utilizzazione schedulabile per RM

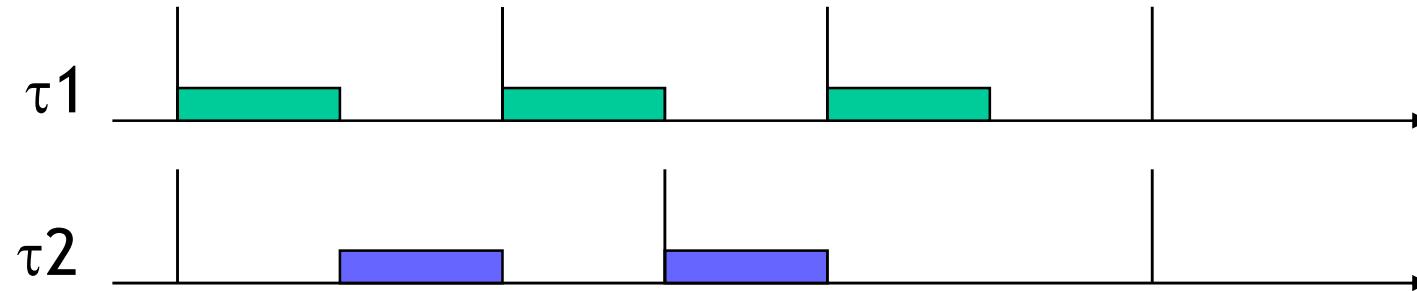
- Teorema: $U_{lub}(RM) < 1$
- Esempio: $\tau_1 = (6, 3)$, $\tau_2 = (9, 4)$
- $U = 3/6 + 4/9 = 0.944 < 1$





Utilizzazione schedulabile per RM

- Esempio: $\tau_1=(6,3)$, $\tau_2=(9,3)$
- $U=3/6+3/9=0.833$

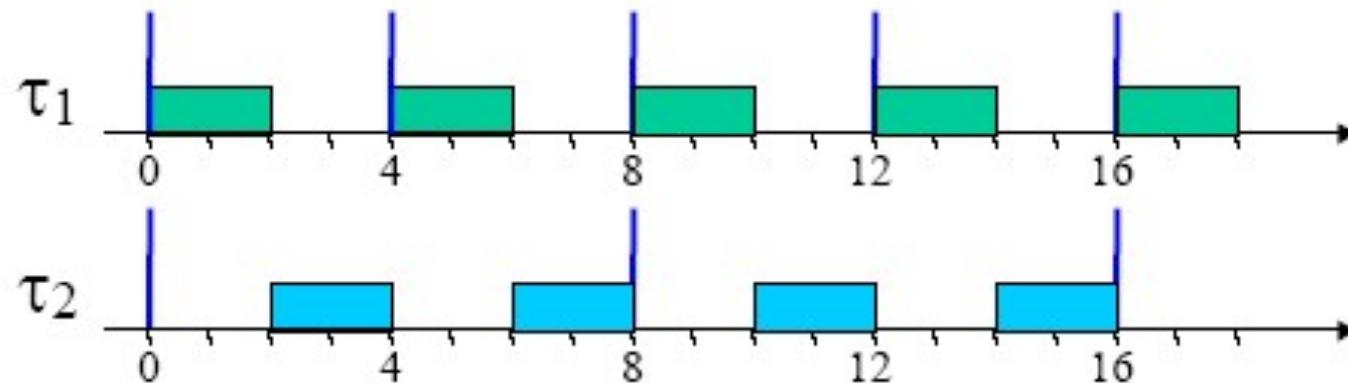


- Nota: se e_1 o e_2 vengono aumentati, τ_2 non rispetterà la deadline
- Situazione di *piena utilizzazione* del processore



Utilizzazione schedulabile per RM

- Esempio: $\tau_1=(4,2)$, $\tau_2=(8,4)$
- $U=1$



- → $U_{ub}(\Gamma)$ dipende non solo dal numero di task, ma anche dai parametri temporali dei task in Γ . U_{lub} ?



Utilizzazione schedulabile per RM

- Teorema (Liu e Layland, 1973)

Least Upper Bound per scheduling RM:

$$U_{lub}(n) = n(2^{1/n} - 1)$$

- Il limite inferiore di $U_{lub}(RM)$ *dipende solo dal numero di task*
- Per $n \rightarrow \infty$ $U_{lub}(RM) \rightarrow \ln 2 \cong 0.693$

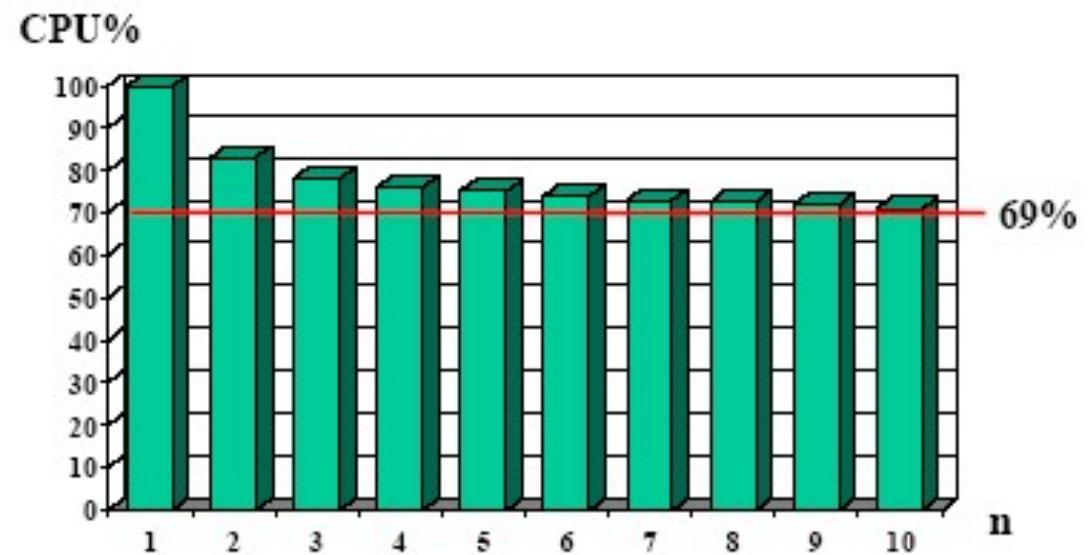


Utilizzazione schedulabile per RM

$$U(n) = n(2^{1/n} - 1)$$

$$U(1)=1.0 \quad U(2)=0.828 \quad U(3)=0.779 \quad U(4)=0.756$$

$$U(5)=0.743 \quad U(6)=0.734 \quad U(7)=0.728 \quad U(8)=0.724$$





Test di Liu e Layland per garanzia RM

- Calcoliamo l'*utilizzazione richiesta* del processore come:

$$U_p = \sum_{i=1}^n \frac{C_i}{T_i}$$

- L'insieme di task è *garantito se* (solo sufficiente):

$$U_p \leq n(2^{1/n} - 1)$$

- E' un test poco costoso, $O(n)$
- Se $n(2^{1/n} - 1) < U_p \leq 1$, l'insieme di task può essere *schedulabile* oppure no



Ipotesi dello scheduling RM

- e_i costante per ogni job di T_i
- p_i costante per ogni job di T_i
- $D_i = p_i$ per ogni task T_i
- Task indipendenti: non ci sono vincoli di precedenza e vincoli su risorse

Ipotesi dello scheduling RM (Liu e Layland, 1973)



1. Le *richieste* di tutti i task con hard deadline sono *periodiche*, con intervalli costanti tra due richieste successive.
2. Per ciascun task il solo vincolo temporale è la sua *completa esecuzione* prima della successiva richiesta di attivazione.
3. I *task* sono *indipendenti*: le richieste di attivazione di un task non dipendono dall'inizio o dal completamento dell'attivazione di altri task.
4. La *durata dell'esecuzione* di ciascun task è *costante* (a meno delle eventuali interruzioni del task).
5. Gli eventuali *task non periodici* sono *non critici* o sono comunque esclusi dall'analisi.



Ottimalità dello scheduling RM

- Teorema: Nelle ipotesi sopracitate, *RM è ottimo tra tutti gli algoritmi a priorità fissa.*
- Se esiste un assegnamento a priorità fissa che genera una schedule fattibile per Γ , anche l'assegnamento RM è fattibile per Γ
- → se Γ non è schedulabile da RM, allora non è schedulabile da alcun assegnamento a priorità fissa, nelle ipotesi date



Istante critico

- *Istante critico* di un task: è l'istante in cui una richiesta di attivazione del task determina il massimo tempo di risposta
- *Intervallo critico* di un task: è il tempo che intercorre tra un istante critico ed il termine della risposta alla richiesta corrispondente

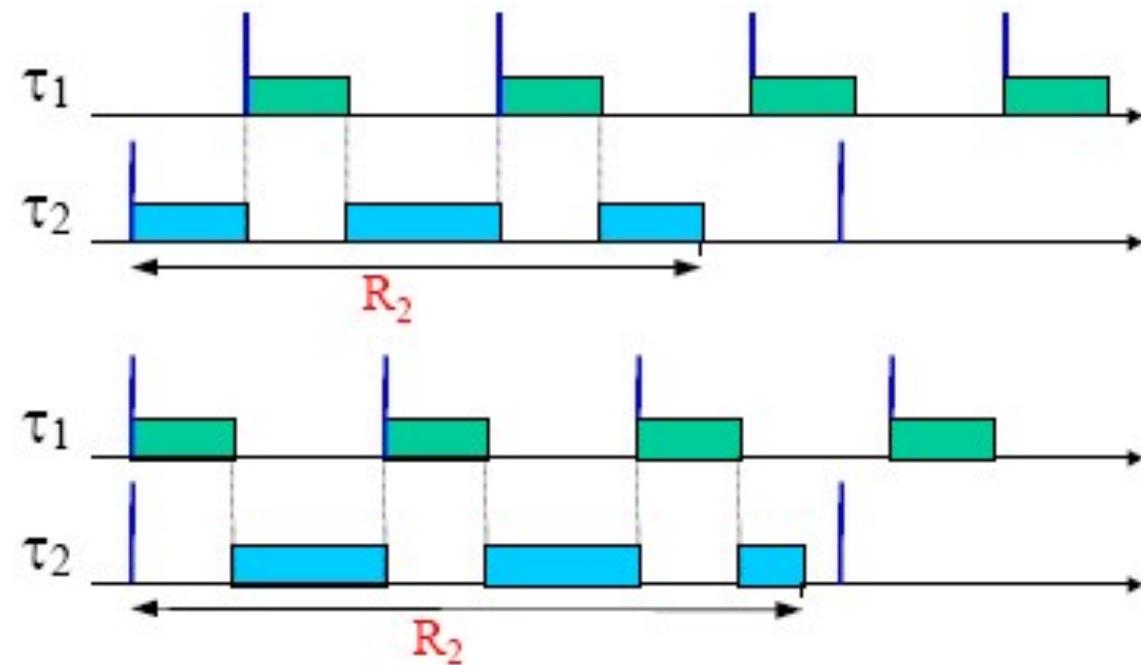
- Fissato l'insieme di task Γ , se un task è schedulabile nel suo intervallo critico (\rightarrow rispetta la sua prima deadline) lo è certamente per qualsiasi relazione di fase tra i task
- Possibile tecnica di analisi: esame degli intervalli critici per tutti i task



Istante critico

- Teorema: In un algoritmo con assegnamento statico delle priorità, per ogni task τ_i il massimo *tempo di risposta* R_i si verifica quando esso viene rilasciato simultaneamente a tutti i task a priorità superiore

- Es. $\text{Pri}(\tau_1) > \text{Pri}(\tau_2)$:





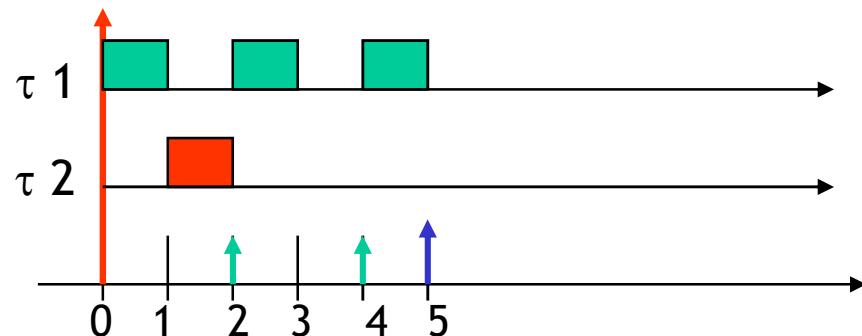
Intervallo critico e analisi di schedulabilità

-
- Un metodo per studiare la schedulabilità di un insieme di task:
 - Analizzare la schedulabilità dei task a priorità inferiore a *partire dall'istante critico*
 - Vale per tutti gli algoritmi di scheduling *statici*
 - Anche non RM e non DM, anche con priorità assegnate in modo arbitrario



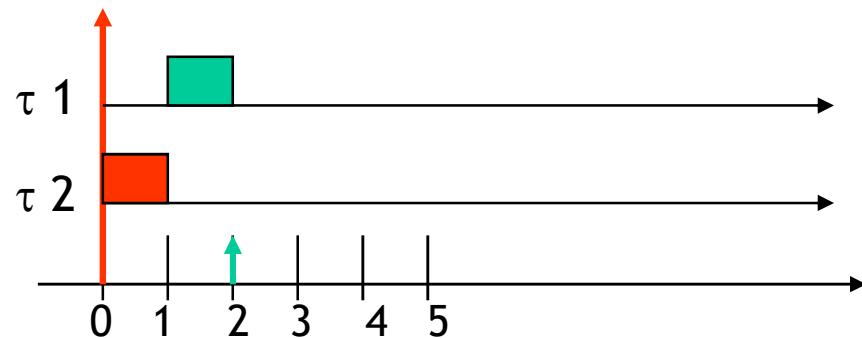
Analisi degli intervalli critici

- Esempio $\tau_1=(2,1)$, $\tau_2=(5,1)$
- Se $\text{Pri}(\tau_1) > \text{Pri}(\tau_2)$ → esame dell'intervallo critico di τ_2 :



- Lo scheduling è fattibile e rimane tale anche incrementando C_2 fino a $C_2=2$

- Se $\text{Pri}(\tau_2) > \text{Pri}(\tau_1)$ → esame dell'intervallo critico di τ_1 :

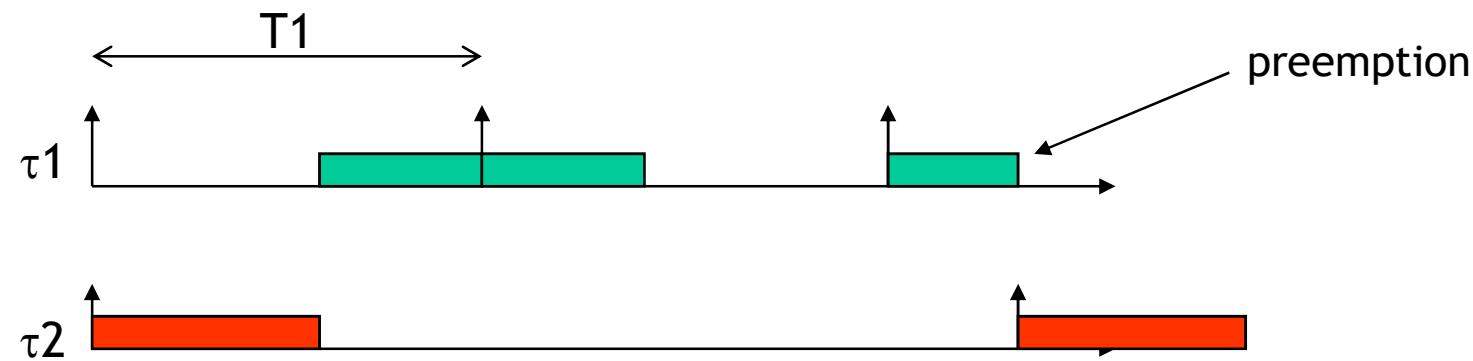


- Lo scheduling è fattibile, ma né C_1 né C_2 possono essere aumentati
- Il processore è *pienamente utilizzato*



Dimostrazione dell'ottimalità di RM

- Consideriamo due task τ_1 e τ_2 con periodi $T_1 < T_2$
- Assegnamo la priorità in modo *non RM*: $\text{Pri}(\tau_2) > \text{Pri}(\tau_1)$
- La schedule è fattibile se: $C_1 + C_2 \leq T_1$ (1)

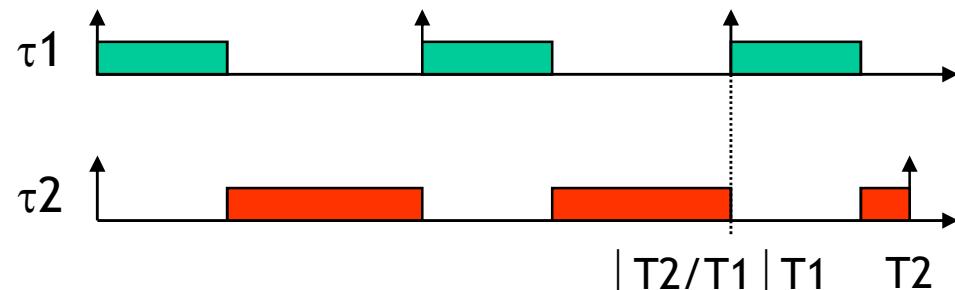


- Il processore è pienamente utilizzato (caso peggiore)



Dimostrazione dell'ottimalità di RM

- Se assegnamo la priorità in modo RM: $\text{Pri}(\tau_1) > \text{Pri}(\tau_2)$
- Si possono verificare due situazioni:
 - (a) Tutte le richieste di τ_1 entro l'intervallo critico di τ_2 sono completate
 - (b) Durante l'esecuzione dell'ultima richiesta di τ_1 si verifica un nuovo rilascio di τ_2

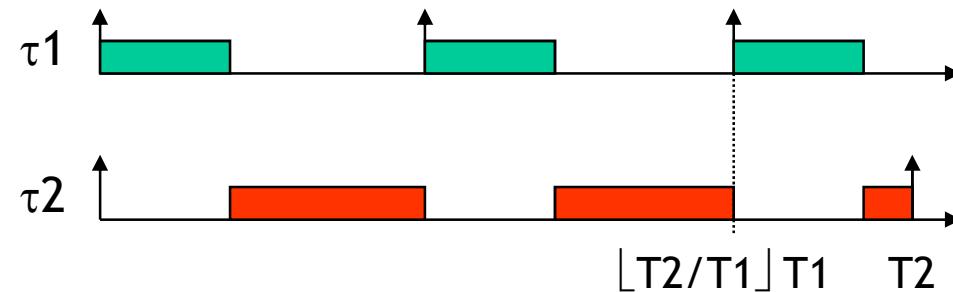


- Caso (a) (def):
$$C_1 \leq T_2 - \lfloor T_2/T_1 \rfloor T_1$$



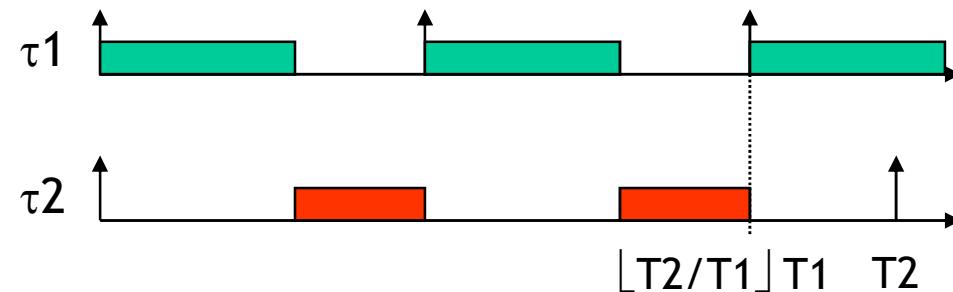
Dimostrazione dell'ottimalità di RM

- Caso a):



$$C_1 \leq T_2 - \lfloor T_2/T_1 \rfloor T_1$$

- Caso b):



$$C_1 \geq T_2 - \lfloor T_2/T_1 \rfloor T_1$$



Dimostrazione dell'ottimalità di RM

- Caso a):

$$C_1 \leq T_2 - \lfloor T_2/T_1 \rfloor T_1$$

- La schedule è fattibile se:

$$(\lfloor T_2/T_1 \rfloor + 1)C_1 + C_2 \leq T_2 \quad (2)$$

- Dimostriamo che se vale la (1), vale anche la (2)

- Moltiplicando la (1) si ha:

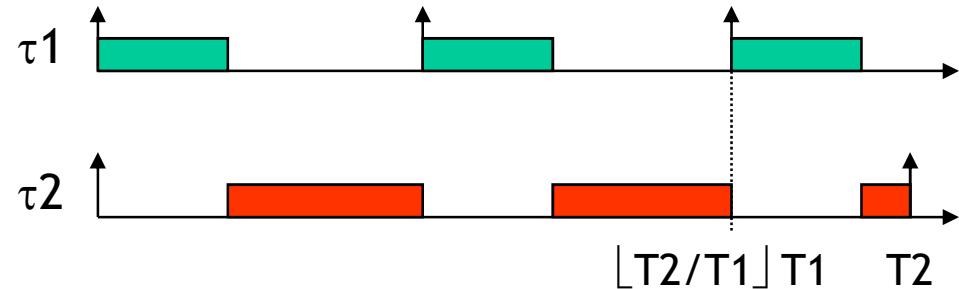
$$\lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor T_1$$

- Essendo $\lfloor T_2/T_1 \rfloor \geq 1$:

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq \lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor T_1$$

- Sommando C_1 ad ogni membro e poichè $C_1 \leq T_2 - \lfloor T_2/T_1 \rfloor T_1$:

$$(\lfloor T_2/T_1 \rfloor + 1) C_1 + C_2 \leq \lfloor T_2/T_1 \rfloor T_1 + C_1 \leq T_2$$

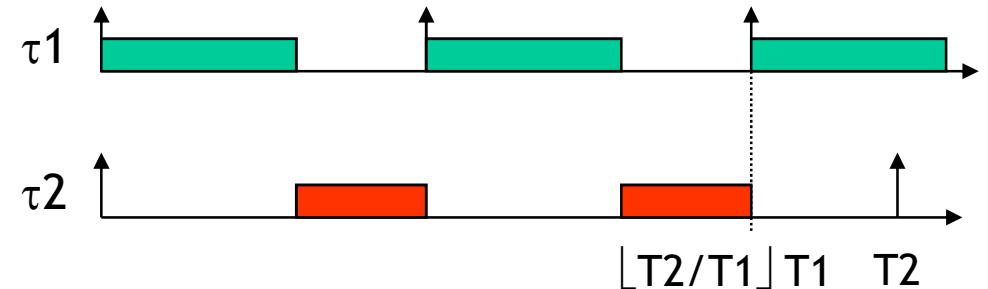




Dimostrazione dell'ottimalità di RM

- **Caso (b):**

$$C_1 \geq T_2 - \lfloor T_2/T_1 \rfloor T_1$$



- La schedule è fattibile se:

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq \lfloor T_2/T_1 \rfloor T_1 \quad (3)$$

- Dimostriamo che se vale la (1), vale anche la (3)

- Moltiplicando la (1) si ha:

$$\lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor T_1$$

- Essendo $\lfloor T_2/T_1 \rfloor \geq 1$:

$$\lfloor T_2/T_1 \rfloor C_1 + C_2 \leq \lfloor T_2/T_1 \rfloor C_1 + \lfloor T_2/T_1 \rfloor C_2 \leq \lfloor T_2/T_1 \rfloor T_1$$



Dimostrazione dell'ottimalità di RM

- Abbiamo dimostrato che se vale (1), vale (2) nel caso (a) e vale (3) nel caso (b) → anche la schedule RM è fattibile
- Estendendo la discussione ad n task e riordinando le priorità tra coppie di task, si dimostra che l'algoritmo RM è ottimo



Applicazione della teoria RM

- Esempio:

$$\tau_1 = (100, 20) \quad U_1 = 0.2$$

$$\tau_2 = (150, 40) \quad U_2 = 0.267$$

$$\tau_3 = (350, 100) \quad U_3 = 0.286$$

- Utilizzazione complessiva richiesta dai 3 task:

$$U = 0.753 \leq U^*(3) = 0.779$$

- I task sono garantiti e schedulabili se lo scheduling è RM, cioè $\text{Pri}(\tau_1) > \text{Pri}(\tau_2) > \text{Pri}(\tau_3)$



Applicazione della teoria RM

- Se $U > U^*(m)$ occorre verificare gli intervalli critici
(notazione: $U^*(m) = U_{lub}(RM)$ per m task)
- Teorema dell'*intervallo critico*: se ogni task rispetta la propria prima deadline quando tutti i task richiedono l'attivazione allo stesso istante, allora tutte le deadline verranno comunque rispettate per qualunque combinazione degli istanti di richiesta
- Modifica dell'esempio precedente con $C_1=40$:

$$U = 0.953 > U^*(3) = 0.779$$



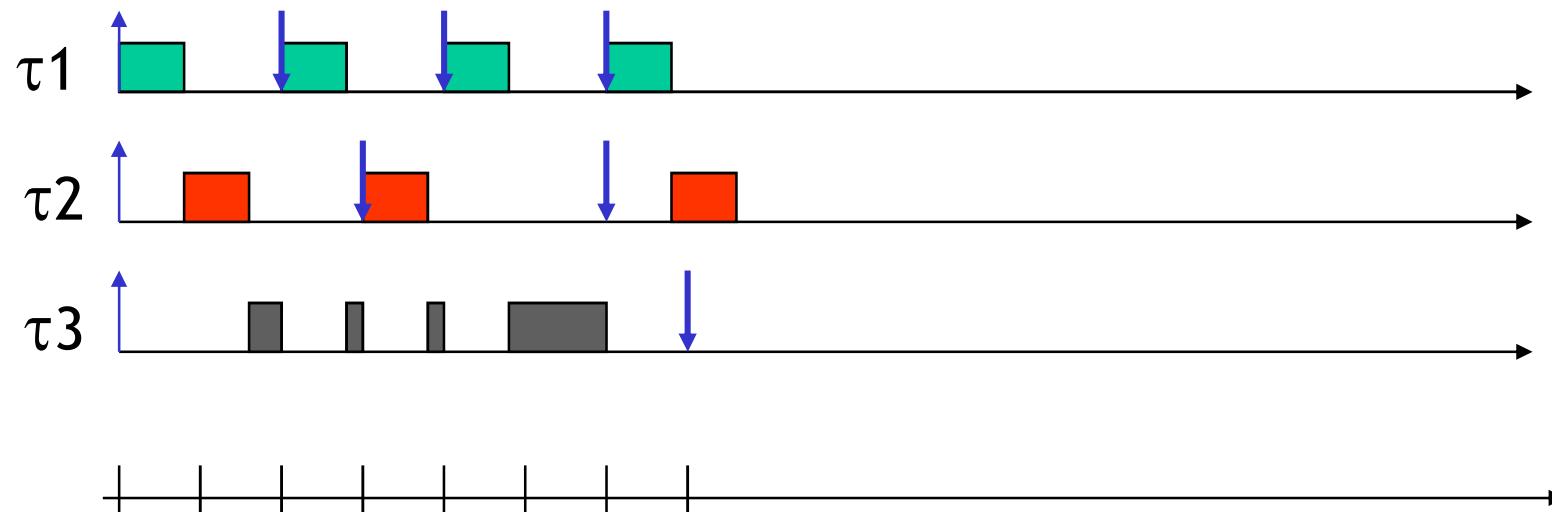
Applicazione della teoria RM

- $\tau_1=(100, 40)$ $\tau_2=(150, 40)$ $\tau_3=(350, 100)$ $U = 0.953$
- τ_1 e τ_2 hanno utilizzazione pari a $0.667 \leq U^*(2)$, pertanto comunque rispettano le proprie deadline
- Per τ_3 si applica il Teorema dell'intervallo critico, verificando se completa la sua esecuzione entro l'intervallo T_3
- La schedulazione statica protegge i task a priorità più elevata; le analisi più onerose sono necessarie solo per i task a priorità più bassa



Applicazione della teoria RM

- In questo caso si verifica che anche τ_3 rispetta la sua deadline
- $\tau_1=(100, 40)$ $\tau_2=(150, 40)$ $\tau_3=(350, 100)$





Considerazioni pratiche

- Utilizzazione schedulabile di RM per Liu e Layland:
$$U_{lub}(RM) \leq n(2^{1/n} - 1)$$
- Per $n \rightarrow \infty$ $U_{lub}(RM) \rightarrow \ln 2 \approx 0.693$
- Se $n(2^{1/n} - 1) < U_p \leq 1$, l'insieme di task può essere schedulabile oppure no
 - task set *schedulabile* secondo il bound LL? \rightarrow forse
 - task set *garantito* dal bound LL? \rightarrow no
- Il bound $U_{lub}(RM)$ di Liu e Layland è un bound *pessimistico*
- Studi su insiemi di task periodici generati casualmente hanno riscontrato un valore probabile del bound di 0.88



Esercizi su scheduling RM con bound Liu-Layland

- Task set in diapositiva n. 41:

$$\tau_1 = (100, 20) \quad \tau_2 = (150, 40) \quad \tau_3 = (350, 100) \quad U = 0.753$$

- I task schedulati in modo RM sono *garantiti* dal bound di Liu-Layland (bound LL) e il task set è *schedulabile*

- Esercizio: Analizzare i seguenti assegnamenti di priorità statica (non RM):

- $\text{pri}(\tau_2) > \text{pri}(\tau_1) > \text{pri}(\tau_3)$
- $\text{pri}(\tau_2) > \text{pri}(\tau_3) > \text{pri}(\tau_1)$
- $\text{pri}(\tau_3) > \text{pri}(\tau_2) > \text{pri}(\tau_1)$
- $\text{pri}(\tau_3) > \text{pri}(\tau_1) > \text{pri}(\tau_2)$
- $\text{pri}(\tau_1) > \text{pri}(\tau_3) > \text{pri}(\tau_2)$

Esercizi su scheduling RM con bound Liu-Layland



- Cosa accade con altri assegnamenti statici di priorità, diversi da RM? Quali tecniche di analisi possiamo utilizzare?

- Per ogni assegnamento di priorità elencato, utilizzando le tecniche e i teoremi visti a lezione, specificare se:
 - ciascun task è individualmente garantito (e perché) -> risposta (sì|no)
 - il task set è schedulabile -> risposta (sì|no|forse)